Proceedings of the 8th Python in Science conference

Gaël Varoquaux, Stefan van Der Walt, Jarrod Millman

SciPy 2009 conference

Python for Scientific Computing

Proceedings of the 8th Python in Science Conference

SciPy Conference – Pasadena, CA, August 18-23, 2009.

Editors: Gaël Varoquaux, Stéfan van der Walt, K. Jarrod Millman

Contents

Proceedings of the 8th Python in Science Conference by Gaël Varoquaux, Stéfan van der Walt, K. Jarrod Milllman
ISBN: 978-0-557-23212-3

Organization

Conference chair

K. Jarrod Millman	UC Berkeley, Helen Wills Neuroscience Institute, USA

Tutorial Co-chairs

Dave Peterson	Enthought Inc, Austin, USA
Fernando Perez	UC Berkeley, Helen Wills Neuroscience Institute, USA

Program Co-chairs

Gaël Varoquaux	INRIA Saclay, France
Stéfan van der Walt	Stellenbosh University, South Africa

Program Committee

Michael Aivazis	Center for Advanced Computing Research, California Institute of Technology USA
Brian Granger	Physics Department, California Polytechnic State University, San Luis Obispo USA
Aric Hagberg	Theoretical Division, Los Alamos National Laboratory USA
Konrad Hinsen	Centre de Biophysique Moléculaire, CNRS Orléans France
Randall LeVeque	Mathematics, University of Washington, Seattle USA
Travis Oliphant	Enthought Inc. USA
Prabhu Ramachandran	Department of Aerospace Engineering, IIT Bombay India
Raphael Ritz	International Neuroinformatics Coordinating Facility Sweden
William Stein	Mathematics, University of Washington, Seattle USA

Proceeding reviewers

Francesc Alted	Pytables Spain
Philippe Ciuciu	CEA, Neurospin France
Yann Cointepas	CEA, Neurospin France
Emmanuelle Gouillart	CNRS Saint Gobain France
Jonathan Guyer	NIST USA
Ben Herbst	Stellenbosh University, South Africa
Paul Kienzle	NIST USA
Michael McKerns	Center for Advanced Computing Research, California Institute of Technology USA
Sturla Molden	University of Oslo Norway
Jean-Baptiste Poline	CEA, Neurospin France
Dag Sverre Seljebotn	University of Oslo Norway
Gregor Thalhammer	University of Florence Italy

Editorial

Gael Varoquaux (gael.varoquaux@normalesup.org) – *INRIA, Saclay* France
Stéfan van der Walt (stefan@sun.ac.za) – *University of Stellenbosch, Stellenbosch* South Africa
Jarrod Millman (millman@berkeley.edu) – *UC Berkeley, Berkeley, CA* USA

SciPy 2009 marks our eighth annual Python in Science conference and the second edition of the conference proceedings. The conference and these proceedings highlight the ongoing focus of the community on providing practical software tools, created to address real scientific problems.

As in previous years, topics at the conference ranged from the presentation of tools and techniques for scientific work with the Python language, to reports on scientific achievement using Python. Interestingly, several people noticed that something important happened in the Scientific Python world during the last year: we are no longer constantly comparing our software with commercial packages, nor justifying the need for Python in Science. Python has now reached the level of adoption where this sort of justification is no longer necessary. The exact moment when this shift in focus occurred is difficult to identify, but that it happened was apparent during the conference.

Recurring scientific themes

This year the conference spanned two days, and each day commenced with a keynote address. The first keynote was delivered by Peter Norvig, the Director of Research at Google; the second by Jonathan Guyer, a materials scientist in the Thermodynamics and Kinetics Group at the National Institute of Standards and Technology (NIST).

Peter Norvig's talk was titled "What to demand from a Scientific Computing Language—even if you don't care about computing or languages", where he discussed a number of desired characteristics in a scientific computing environment. Such a platform should have the ability to share code and data with other researchers easily, provide extremely fast computations and state-of-the-art algorithms to researchers in the field, and be as easy as possible to use in a time-efficient manner. He also stressed the importance of having code that read like the mathematical ideas it expressed.

Jonathan Guyer's keynote centred around "Modeling of Materials with Python". He expanded on several of the above-mentioned characteristics as he discussed the development of FiPy, a framework for solving partial differential equations based on a finite volume approach. Jonathan explained how FiPy was created to provide the most advanced numerical techniques to scientists, so that they could focus on the scientific questions at hand, while having a standard platform for sharing codes with colleagues. Importantly, FiPy has become a critical tool in Jonathan's research group and has been adopted by many of their colleagues for both research as well as teaching.

Both keynote addresses served to outline prominent themes that were repeated throughout the conference, as witnessed by the proceedings. These themes include: the need for software tools that allow scientists to focus on their research, while taking advantage of best-of-class algorithms and utilizing the full power of their computational resources; the need for a high-level computing environment with an easy—to-write and read syntax; the usefulness of high-quality software tools for teaching and education; and the importance of sharing code and data in scientific research.

The first several articles address high-level approaches aimed at improving the performance of numerical code written in Python. While making better use of increased computation resources, such as parallel processors or graphical processing units, many of these approaches also focus on reducing code complexity and verbosity. Again, simpler software allows scientists to focus on the details of their computations, rather than on administrating their computing resources.

The remaining articles focus on work done to solve problems in specific research domains, ranging from numerical methods to biology and astronomy. For the last several years, using Python to wrap existing libraries has been a popular way to provide a scripting frontend to computational code written primarily in a more low-level language like C or Fortran. However, as these proceedings show, Python is increasingly used as the primary language for large scientific applications. Python and its stack of scientific tools appears to be well suited for application areas ranging from database applications to user interfaces and numerical computation.

Review and selection process

This year we received 30 abstracts from five different countries. The submissions covered a number of research fields, including bioinformatics, computer vision, nanomaterials, neutron scattering, neuroscience, applied mathematics, astronomy, and X-ray fluorescence. Moreover, the articles discussed involve a number of computational tools: these include statistical modeling, data mining, visualization, performance optimization, parallel computing, code wrapping, instrument control, time series analysis, geographic information science, spatial data analysis, adaptive interpolation, spectral analysis, symbolic mathematics, finite element, and virtual reality. Several abstracts also addressed the role of scientific Python in teaching and education.

Each abstract was reviewed by both the program chairs, as well as two members of the program committee (PC). The PC consisted of 11 members from five countries, and represented both industry and academia. Abstracts were evaluated according to the following criteria:

- Relevance of the contribution, with regard to the topics and goals of the conference.
- Scientific or technical quality of the work presented.
- Originality and soundness.

We accepted 23 (76%) submission for oral presentation at the conference. At the closure of the conference, we invited the presenters to submit their work for publication in the conference proceedings. These submissions were reviewed by 11 proceeding reviewers from seven countries, according to the following criteria:

- Does the paper describe a well-formulated scientific or technical achievement?
- Is the content of the paper accessible to a computational scientist with no specific knowledge in the given field?
- Are the technical and scientific decisions well-motivated?
- Does the paper reference scientific sources and material used?
- Are the code examples (if any) sound, clear, and well-written?
- Is the paper fit for publication in the SciPy proceedings? Improvements may be suggested, with or without a second review.

From the 30 original abstracts, 12(40%) have been accepted for publication in these proceedings.

Prior to commencing the conference, we had two days of tutorials with both an introductory and advanced track. In addition to publishing a selection of the presented work, we also selected one of this year's tutorial presentations for publication.

The proceedings conclude with a short progress report on the two-year long NumPy and SciPy documentation project.

The SciPy Conference has been supported since its inception by the Center for Advanced Computing Research (CACR) at Caltech and Enthought Inc. In addition, we were delighted to receive funding this year from the Python Software Foundation to cover the travel, registration, and accommodation expenses of 10 students. Finally, we are very grateful to Leah Jones of Enthought and Julie Ponce of the CACR for their invaluable help in organizing the conference.

Cython tutorial

Stefan Behnel (stefan_ml@behnel.de) – *Senacor Technologies AG*, Germany
Robert W. Bradshaw (robertwb@math.washington.edu) – *University of Washington*[a], USA
Dag Sverre Seljebotn (dagss@student.matnat.uio.no) – *University of Oslo*[bcd], Norway

[a]Department of Mathematics, University of Washington, Seattle, WA, USA
[b]Institute of Theoretical Astrophysics, University of Oslo, P.O. Box 1029 Blindern, N-0315 Oslo, Norway
[c]Department of Mathematics, University of Oslo, P.O. Box 1053 Blindern, N-0316 Oslo, Norway
[d]Centre of Mathematics for Applications, University of Oslo, P.O. Box 1053 Blindern, N-0316 Oslo, Norway

Cython is a programming language based on Python with extra syntax to provide static type declarations. This takes advantage of the benefits of Python while allowing one to achieve the speed of C. In this paper we describe the Cython language and show how it can be used both to write optimized code and to interface with external C libraries.

Cython - an overview

[Cython] is a programming language based on Python, with extra syntax allowing for optional static type declarations. It aims to become a superset of the [Python] language which gives it high-level, object-oriented, functional, and dynamic programming. The source code gets translated into optimized C/C++ code and compiled as Python extension modules. This allows for both very fast program execution and tight integration with external C libraries, while keeping up the high programmer productivity for which the Python language is well known.

The primary Python execution environment is commonly referred to as CPython, as it is written in C. Other major implementations use Java (Jython [Jython]), C# (IronPython [IronPython]) and Python itself (PyPy [PyPy]). Written in C, CPython has been conducive to wrapping many external libraries that interface through the C language. It has, however, remained non trivial to write the necessary glue code in C, especially for programmers who are more fluent in a high-level language like Python than in a do-it-yourself language like C.

Originally based on the well-known Pyrex [Pyrex], the Cython project has approached this problem by means of a source code compiler that translates Python code to equivalent C code. This code is executed within the CPython runtime environment, but at the speed of compiled C and with the ability to call directly into C libraries. At the same time, it keeps the original interface of the Python source code, which makes it directly usable from Python code. These two-fold characteristics enable Cython's two major use cases: extending the CPython interpreter with fast binary modules, and interfacing Python code with external C libraries.

While Cython can compile (most) regular Python code, the generated C code usually gains major (and sometime impressive) speed improvements from optional static type declarations for both Python and C types. These allow Cython to assign C semantics to parts of the code, and to translate them into very efficient C code. Type declarations can therefore be used for two purposes: for moving code sections from dynamic Python semantics into static-and-fast C semantics, but also for directly manipulating types defined in external libraries. Cython thus merges the two worlds into a very broadly applicable programming language.

Installing Cython

Many scientific Python distributions, such as the Enthought Python Distribution [EPD], Python(x,y) [Pythonxy], and Sage [Sage], bundle Cython and no setup is needed. Note however that if your distribution ships a version of Cython which is too old you can still use the instructions below to update Cython. Everything in this tutorial should work with Cython 0.11.2 and newer, unless a footnote says otherwise.

Unlike most Python software, Cython requires a C compiler to be present on the system. The details of getting a C compiler varies according to the system used:

- **Linux** The GNU C Compiler (gcc) is usually present, or easily available through the package system. On Ubuntu or Debian, for instance, the command `sudo apt-get install build-essential` will fetch everything you need.
- **Mac OS X** To retrieve gcc, one option is to install Apple's XCode, which can be retrieved from the Mac OS X's install DVDs or from http://developer.apple.com.
- **Windows** A popular option is to use the open source MinGW (a Windows distribution of gcc). See the appendix for instructions for setting up MinGW manually. EPD and Python(x,y) bundle MinGW, but some of the configuration steps in the appendix might still be necessary. Another option is to use Microsoft's Visual C. One must then use the same version which the installed Python was compiled with.

The newest Cython release can always be downloaded from http://cython.org. Unpack the tarball or zip file, enter the directory, and then run:

```
python setup.py install
```

If you have Python setuptools set up on your system, you should be able to fetch Cython from PyPI and install it using:

```
easy_install cython
```

For Windows there is also an executable installer available for download.

Building Cython code

Cython code must, unlike Python, be compiled. This happens in two stages:

- A `.pyx` file is compiled by Cython to a `.c` file, containing the code of a Python extension module
- The `.c` file is compiled by a C compiler to a `.so` file (or `.pyd` on Windows) which can be `import`-ed directly into a Python session.

There are several ways to build Cython code:

- Write a distutils `setup.py`.
- Use `pyximport`, importing Cython `.pyx` files as if they were `.py` files (using distutils to compile and build the background).
- Run the `cython` command-line utility manually to produce the `.c` file from the `.pyx` file, then manually compiling the `.c` file into a shared object library or `.dll` suitable for import from Python. (This is mostly for debugging and experimentation.)
- Use the [Sage] notebook which allows Cython code inline and makes it easy to experiment with Cython code without worrying about compilation details (see figure 1 below).

Currently, distutils is the most common way Cython files are built and distributed.

Building a Cython module using distutils

Imagine a simple "hello world" script in a file `hello.pyx`:

```
def say_hello_to(name):
    print(Hello %s!" % name)
```

The following could be a corresponding `setup.py` script:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [Extension("hello", ["hello.pyx"])]

setup(
  name = 'Hello world app',
  cmdclass = {'build_ext': build_ext},
  ext_modules = ext_modules
)
```

To build, run `python setup.py build_ext --inplace`. Then simply start a Python session and do `from hello import say_hello_to` and use the imported function as you see fit.

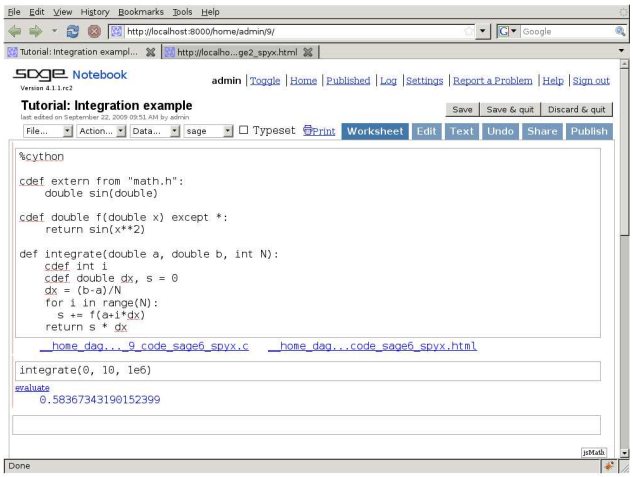


Figure 1 *The Sage notebook allows transparently editing and compiling Cython code simply by typing* `%cython` *at the top of a cell and evaluate it. Variables and functions defined in a Cython cell imported into the running session.*

Data types in Cython

Cython is a Python compiler. This means that it can compile normal Python code without changes (with a few obvious exceptions of some as-yet unsupported language features). However, for performance-critical code, it is often helpful to add static type declarations, as they will allow Cython to step out of the dynamic nature of the Python code and generate simpler and faster C code - sometimes faster by orders of magnitude.

It must be noted, however, that type declarations can make the source code more verbose and thus less readable. It is therefore discouraged to use them without good reason, such as where benchmarks prove that they really make the code substantially faster in a performance critical section. Typically a few types in the right spots go a long way. Cython can produce annotated output (see figure 2 below) that can be very useful in determining where to add types.

All C types are available for type declarations: integer and floating point types, complex numbers, structs, unions and pointer types. Cython can automatically and correctly convert between the types on assignment. This also includes Python's arbitrary size integer types, where value overflows on conversion to a C type will raise a Python `OverflowError` at runtime. The generated C code will handle the platform dependent sizes of C types correctly and safely in this case.

Faster code by adding types

Consider the following pure Python code:

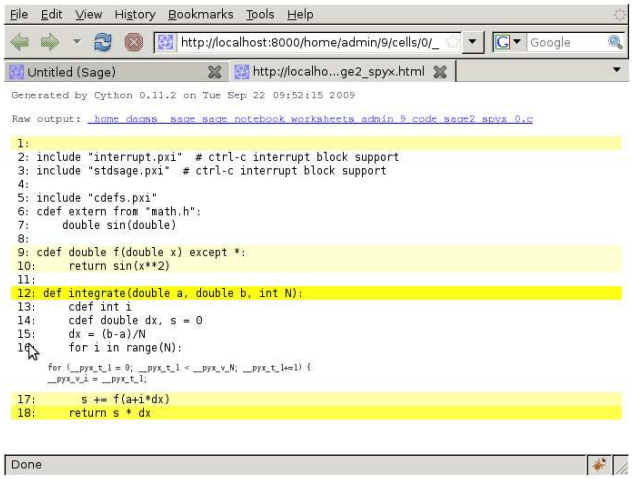


Figure 2 *Using the* `-a` *switch to the* `cython` *command line program (or following a link from the Sage notebook) results in an HTML report of Cython code interleaved with the generated C code. Lines are colored according to the level of "typedness" - white lines translates to pure C without any Python API calls. This report is invaluable when optimizing a function for speed.*

```
from math import sin

def f(x):
    return sin(x**2)

def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

Simply compiling this in Cython merely gives a 5% speedup. This is better than nothing, but adding some static types can make a much larger difference.

With additional type declarations, this might look like:

```
from math import sin

def f(double x):
    return sin(x**2)

def integrate_f(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

Since the iterator variable `i` is typed with C semantics, the for-loop will be compiled to pure C code. Typing `a`, `s` and `dx` is important as they are involved in arithmetic withing the for-loop; typing `b` and `N` makes less of a difference, but in this case it is not much extra work to be consistent and type the entire function.

This results in a 24 times speedup over the pure Python version.

cdef functions

Python function calls can be expensive, and this is especially true in Cython because one might need to convert to and from Python objects to do the call. In our example above, the argument is assumed to be a C double both inside f() and in the call to it, yet a Python `float` object must be constructed around the argument in order to pass it.

Therefore Cython provides a syntax for declaring a C-style function, the cdef keyword:

```
cdef double f(double) except *:
    return sin(x**2)
```

Some form of except-modifier should usually be added, otherwise Cython will not be able to propagate exceptions raised in the function (or a function it calls). Above `except *` is used which is always safe. An except clause can be left out if the function returns a Python object or if it is guaranteed that an exception will not be raised within the function call.

A side-effect of cdef is that the function is no longer available from Python-space, as Python wouldn't know how to call it. Using the `cpdef` keyword instead of cdef, a Python wrapper is also created, so that the function is available both from Cython (fast, passing typed values directly) and from Python (wrapping values in Python objects).

Note also that it is no longer possible to change `f` at runtime.

Speedup: 45 times over pure Python.

Calling external C functions

It is perfectly OK to do `from math import sin` to use Python's `sin()` function. However, calling C's own `sin()` function is substantially faster, especially in tight loops. It can be declared and used in Cython as follows:

```
cdef extern from "math.h":
    double sin(double)

cdef double f(double x):
    return sin(x*x)
```

At this point there are no longer any Python wrapper objects around our values inside of the main for loop, and so we get an impressive speedup to 219 times the speed of Python.

Note that the above code re-declares the function from `math.h` to make it available to Cython code. The C compiler will see the original declaration in `math.h` at compile time, but Cython does not parse "math.h" and requires a separate definition.

When calling C functions, one must take care to link in the appropriate libraries. This can be platform-specific; the below example works on Linux and Mac OS X:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules=[
    Extension("demo",
              ["demo.pyx"],
              libraries=["m"]) # Unix-like specific
]

setup(
  name = "Demos",
  cmdclass = {"build_ext": build_ext},
  ext_modules = ext_modules
)
```

If one uses the Sage notebook to compile Cython code, one can use a special comment to tell Sage to link in libraries:

```
#clib: m
```

Just like the `sin()` function from the math library, it is possible to declare and call into any C library as long as the module that Cython generates is properly linked against the shared or static library. A more extensive example of wrapping a C library is given in the section Using C libraries.

Extension types (aka. cdef classes)

To support object-oriented programming, Cython supports writing normal Python classes exactly as in Python:

```
class MathFunction(object):
    def __init__(self, name, operator):
        self.name = name
        self.operator = operator

    def __call__(self, *operands):
        return self.operator(*operands)
```

Based on what Python calls a "built-in type", however, Cython supports a second kind of class: *extension types*, sometimes referred to as "cdef classes" due to the keywords used for their declaration. They are somewhat restricted compared to Python classes, but are generally more memory efficient and faster than generic Python classes. The main difference is that they use a C struct to store their fields and methods instead of a Python dict. This allows them to store arbitrary C types in their fields without requiring a Python wrapper for them, and to access fields and methods directly at the C level without passing through a Python dictionary lookup.

Normal Python classes can inherit from cdef classes, but not the other way around. Cython requires to know the complete inheritance hierarchy in order to lay out their C structs, and restricts it to single inheritance. Normal Python classes, on the other hand, can inherit from any number of Python classes and extension types, both in Cython code and pure Python code.

So far our integration example has not been very useful as it only integrates a single hard-coded function. In order to remedy this, without sacrificing speed, we will use a cdef class to represent a function on floating point numbers:

```
cdef class Function:
    cpdef double evaluate(self, double x) except *:
        return 0
```

Like before, cpdef makes two versions of the method available; one fast for use from Cython and one slower for use from Python. Then:

```
cdef class SinOfSquareFunction(Function):
    cpdef double evaluate(self, double x) except *:
        return sin(x**2)
```

Using this, we can now change our integration example:

```
def integrate(Function f, double a, double b, int N):
  cdef int i
  cdef double s, dx
  if f is None:
      raise ValueError("f cannot be None")
  s = 0
  dx = (b-a)/N
  for i in range(N):
      s += f.evaluate(a+i*dx)
  return s * dx

print(integrate(SinOfSquareFunction(), 0, 1, 10000))
```

This is almost as fast as the previous code, however it is much more flexible as the function to integrate can be changed. It is even possible to pass in a new function defined in Python-space. Assuming the above code is in the module *integrate.pyx*, we can do:

```
>>> import integrate
>>> class MyPolynomial(integrate.Function):
...     def evaluate(self, x):
...         return 2*x*x + 3*x - 10
...
>>> integrate.integrate(MyPolynomial(), 0, 1, 10000)
-7.8335833300000077
```

This is about 20 times slower than *SinOfSquareFunction*, but still about 10 times faster than the original Python-only integration code. This shows how large the speed-ups can easily be when whole loops are moved from Python code into a Cython module.

Some notes on our new implementation of `evaluate`:

- The fast method dispatch here only works because `evaluate` was declared in `Function`. Had `evaluate` been introduced in `SinOfSquareFunction`, the code would still work, but Cython would have used the slower Python method dispatch mechanism instead.
- In the same way, had the argument `f` not been typed, but only been passed as a Python object, the slower Python dispatch would be used.
- Since the argument is typed, we need to check whether it is `None`. In Python, this would have resulted in an `AttributeError` when the `evaluate` method was looked up, but Cython would instead try to access the (incompatible) internal structure of `None` as if it were a `Function`, leading to a crash or data corruption.

There is a *compiler directive* `nonecheck` which turns on checks for this, at the cost of decreased speed. Here's how compiler directives are used to dynamically switch on or off `nonecheck`:

```
#cython: nonecheck=True
#        ^^^ Turns on nonecheck globally

import cython

# Turn off nonecheck locally for the function
@cython.nonecheck(False)
def func():
    cdef MyClass obj = None
    try:
        # Turn nonecheck on again for a block
        with cython.nonecheck(True):
            print obj.myfunc() # Raises exception
    except AttributeError:
        pass
    print obj.myfunc() # Hope for a crash!
```

Attributes in cdef classes behave differently from attributes in regular classes:

- All attributes must be pre-declared at compile-time
- Attributes are by default only accessible from Cython (typed access)
- Properties can be declared to expose dynamic attributes to Python-space

```
cdef class WaveFunction(Function):
    # Not available in Python-space:
    cdef double offset
    # Available in Python-space:
    cdef public double freq
    # Available in Python-space:
    property period:
        def __get__(self):
            return 1.0 / self. freq
        def __set__(self, value):
            self. freq = 1.0 / value
    <...>
```

pxd files

In addition to the `.pyx` source files, Cython uses `.pxd` files which work like C header files - they contain Cython declarations (and sometimes code sections) which are only meant for sharing C-level declarations with other Cython modules. A `pxd` file is imported into a `pyx` module by using the `cimport` keyword.
`pxd` files have many use-cases:

1. They can be used for sharing external C declarations.
2. They can contain functions which are well suited for inlining by the C compiler. Such functions should be marked `inline`, example:

   ```
   cdef inline int int_min(int a, int b):
       return b if b < a else a
   ```

3. When accompanying an equally named `pyx` file, they provide a Cython interface to the Cython module so that other Cython modules can communicate with it using a more efficient protocol than the Python one.

In our integration example, we might break it up into `pxd` files like this:

1. Add a `cmath.pxd` function which defines the C functions available from the C `math.h` header file, like `sin`. Then one would simply do `from cmath import sin` in `integrate.pyx`.
2. Add a `integrate.pxd` so that other modules written in Cython can define fast custom functions to integrate.

   ```
   cdef class Function:
       cpdef evaluate(self, double x)
   cpdef integrate(Function f, double a,
                   double b, int N)
   ```

   Note that if you have a cdef class with attributes, the attributes must be declared in the class declaration `pxd` file (if you use one), not the `pyx` file. The compiler will tell you about this.

Using Cython with NumPy

Cython has support for fast access to NumPy arrays. To optimize code using such arrays one must `cimport` the NumPy pxd file (which ships with Cython), and declare any arrays as having the `ndarray` type. The data type and number of dimensions should be fixed at compile-time and passed. For instance:

```
import numpy as np
cimport numpy as np
def myfunc(np.ndarray[np.float64_t, ndim=2] A):
    <...>
```

`myfunc` can now only be passed two-dimensional arrays containing double precision floats, but array indexing operation is much, much faster, making it suitable for numerical loops. Expect speed increases well over 100 times over a pure Python loop; in some cases the speed increase can be as high as 700 times or more. [Seljebotn09] contains detailed examples and benchmarks.
Fast array declarations can currently only be used with function local variables and arguments to `def`-style functions (not with arguments to `cpdef` or `cdef`, and neither with fields in cdef classes or as global variables). These limitations are considered known defects and we hope to remove them eventually. In most circumstances it is possible to work around these limitations rather easily and without a significant speed penalty, as all NumPy arrays can also be passed as untyped objects.
Array indexing is only optimized if exactly as many indices are provided as the number of array dimensions. Furthermore, all indices must have a native integer type. Slices and NumPy "fancy indexing" is not optimized. Examples:


```
def myfunc(np.ndarray[np.float64_t, ndim=1] A):
    cdef Py_ssize_t i, j
    for i in range(A.shape[0]):
        print A[i, 0] # fast
        j = 2*i
        print A[i, j] # fast
        k = 2*i
        print A[i, k] # slow, k is not typed
        print A[i][j] # slow
        print A[i,:]  # slow
```

`Py_ssize_t` is a signed integer type provided by Python which covers the same range of values as is supported as NumPy array indices. It is the preferred type to use for loops over arrays.

Any Cython primitive type (float, complex float and integer types) can be passed as the array data type. For each valid dtype in the `numpy` module (such as `np.uint8`, `np.complex128`) there is a corresponding Cython compile-time definition in the cimport-ed NumPy pxd file with a `_t` suffix[1]. Cython structs are also allowed and corresponds to NumPy record arrays. Examples:

```
cdef packed struct Point:
    np.float64_t x, y

def f():
    cdef np.ndarray[np.complex128_t, ndim=3] a = \
        np.zeros((3,3,3), dtype=np.complex128)
    cdef np.ndarray[Point] b = np.zeros(10,
        dtype=np.dtype([('x', np.float64),
                        ('y', np.float64)]))
    <...>
```

Note that `ndim` defaults to 1. Also note that NumPy record arrays are by default unaligned, meaning data is packed as tightly as possible without considering the alignment preferences of the CPU. Such unaligned record arrays corresponds to a Cython `packed` struct. If one uses an aligned dtype, by passing `align=True` to the `dtype` constructor, one must drop the `packed` keyword on the struct definition.

Some data types are not yet supported, like boolean arrays and string arrays. Also data types describing data which is not in the native endian will likely never be supported. It is however possible to access such arrays on a lower level by casting the arrays:

```
cdef np.ndarray[np.uint8, cast=True] boolarr = (x < y)
cdef np.ndarray[np.uint32, cast=True] values = \
    np.arange(10, dtype='>i4')
```

Assuming one is on a little-endian system, the `values` array can still access the raw bit content of the array (which must then be reinterpreted to yield valid results on a little-endian system).

Finally, note that typed NumPy array variables in some respects behave a little differently from untyped arrays. `arr.shape` is no longer a tuple. `arr.shape[0]` is valid but to e.g. print the shape one must do `print (<object>arr).shape` in order to "untype" the variable first. The same is true for `arr.data` (which in typed mode is a C data pointer).

There are many more options for optimizations to consider for Cython and NumPy arrays. We again refer the interested reader to [Seljebotn09].

[1] In Cython 0.11.2, `np.complex64_t` and `np.complex128_t` does not work and one must write `complex` or `double complex` instead. This is fixed in 0.11.3. Cython 0.11.1 and earlier does not support complex numbers.

Using C libraries

Apart from writing fast code, one of the main use cases of Cython is to call external C libraries from Python code. As seen for the C string decoding functions above, it is actually trivial to call C functions directly in the code. The following describes what needs to be done to use an external C library in Cython code.

Imagine you need an efficient way to store integer values in a FIFO queue. Since memory really matters, and the values are actually coming from C code, you cannot afford to create and store Python `int` objects in a list or deque. So you look out for a queue implementation in C.

After some web search, you find the C-algorithms library [CAlg] and decide to use its double ended queue implementation. To make the handling easier, however, you decide to wrap it in a Python extension type that can encapsulate all memory management.

The C API of the queue implementation, which is defined in the header file `libcalg/queue.h`, essentially looks like this:

```
typedef struct _Queue Queue;
typedef void *QueueValue;

Queue *queue_new(void);
void queue_free(Queue *queue);

int queue_push_head(Queue *queue, QueueValue data);
QueueValue queue_pop_head(Queue *queue);
QueueValue queue_peek_head(Queue *queue);

int queue_push_tail(Queue *queue, QueueValue data);
QueueValue queue_pop_tail(Queue *queue);
QueueValue queue_peek_tail(Queue *queue);

int queue_is_empty(Queue *queue);
```

To get started, the first step is to redefine the C API in a `.pxd` file, say, `cqueue.pxd`:

```
cdef extern from "libcalg/queue.h":
    ctypedef struct Queue:
        pass
    ctypedef void* QueueValue

    Queue* new_queue()
    void queue_free(Queue* queue)

    int queue_push_head(Queue* queue, QueueValue data)
    QueueValue  queue_pop_head(Queue* queue)
    QueueValue queue_peek_head(Queue* queue)

    int queue_push_tail(Queue* queue, QueueValue data)
    QueueValue queue_pop_tail(Queue* queue)
    QueueValue queue_peek_tail(Queue* queue)

    bint queue_is_empty(Queue* queue)
```

Note how these declarations are almost identical to the header file declarations, so you can often just copy them over. One exception is the last line. The return value of the `queue_is_empty` method is actually a C boolean value, i.e. it is either zero or non-zero, indicating if the queue is empty or not. This is best expressed

by Cython's `bint` type, which is a normal `int` type when used in C but maps to Python's boolean values `True` and `False` when converted to a Python object. Another difference is the first line. `Queue` is in this case used as an *opaque handle*; only the library that is called know what is actually inside. Since no Cython code needs to know the contents of the struct, we do not need to declare its contents, so we simply provide an empty definition (as we do not want to declare the `_Queue` type which is referenced in the C header)[2].

Next, we need to design the Queue class that should wrap the C queue. Here is a first start for the Queue class:

```
cimport cqueue
cimport python_exc

cdef class Queue:
    cdef cqueue.Queue _c_queue
    def __cinit__(self):
        self._c_queue = cqueue.new_queue()
```

Note that it says `__cinit__` rather than `__init__`. While `__init__` is available as well, it is not guaranteed to be run (for instance, one could create a subclass and forget to call the ancestor constructor). Because not initializing C pointers often leads to crashing the Python interpreter without leaving as much as a stack trace, Cython provides `__cinit__` which is *always* called on construction. However, as `__cinit__` is called during object construction, `self` is not fully constructed yet, and one must avoid doing anything with `self` but assigning to `cdef` fields.

Note also that the above method takes no parameters, although subtypes may want to accept some. Although it is guaranteed to get called, the no-arguments `__cinit__()` method is a special case here as it does not prevent subclasses from adding parameters as they see fit. If parameters are added they must match those of any declared `__init__` method.

Before we continue implementing the other methods, it is important to understand that the above implementation is not safe. In case anything goes wrong in the call to `new_queue()`, this code will simply swallow the error, so we will likely run into a crash later on. According to the documentation of the `new_queue()` function, the only reason why the above can fail is due to insufficient memory. In that case, it will return `NULL`, whereas it would normally return a pointer to the new queue.

The normal way to get out of this is to raise an exception, but allocating a new exception instance may actually fail when we are running out of memory. Luckily, CPython provides a function `PyErr_NoMemory()` that raises the right exception for us. We can thus change the init function as follows:

[2]There's a subtle difference between `cdef struct Queue: pass` and `ctypedef struct Queue: pass`. The former declares a type which is referenced in C code as `struct Queue`, while the latter is referenced in C as `Queue`. This is a C language quirk that Cython is not able to hide. Most modern C libraries use the `ctypedef` kind of struct.

```
def __cinit__(self):
    self._c_queue = cqueue.new_queue()
    if self._c_queue is NULL:
        python_exc.PyErr_NoMemory()
```

The next thing to do is to clean up when the Queue is no longer used. To this end, CPython provides a callback that Cython makes available as a special method `__dealloc__()`. In our case, all we have to do is to free the Queue, but only if we succeeded in initialising it in the init method:

```
def __dealloc__(self):
    if self._c_queue is not NULL:
        cqueue.queue_free(self._c_queue)
```

At this point, we have a compilable Cython module that we can test. To compile it, we need to configure a `setup.py` script for distutils. Based on the example presented earlier on, we can extend the script to include the necessary setup for building against the external C library. Assuming it's installed in the normal places (e.g. under `/usr/lib` and `/usr/include` on a Unix-like system), we could simply change the extension setup from

```
ext_modules = [Extension("hello", ["hello.pyx"])]
```

to

```
ext_modules = [
    Extension("hello", ["hello.pyx"],
              libraries=["calg"])
    ]
```

If it is not installed in a 'normal' location, users can provide the required parameters externally by passing appropriate C compiler flags, such as:

```
CFLAGS="-I/usr/local/otherdir/calg/include"  \
LDFLAGS="-L/usr/local/otherdir/calg/lib"     \
    python setup.py build_ext -i
```

Once we have compiled the module for the first time, we can try to import it:

```
PYTHONPATH=. python -c 'import queue.Queue as Q; Q()'
```

However, our class doesn't do much yet so far, so let's make it more usable.

Before implementing the public interface of this class, it is good practice to look at what interfaces Python offers, e.g. in its `list` or `collections.deque` classes. Since we only need a FIFO queue, it's enough to provide the methods `append()`, `peek()` and `pop()`, and additionally an `extend()` method to add multiple values at once. Also, since we already know that all values will be coming from C, it's better to provide only `cdef` methods for now, and to give them a straight C interface.

In C, it is common for data structures to store data as a `void*` to whatever data item type. Since we only want to store `int` values, which usually fit into the size of a pointer type, we can avoid additional memory allocations through a trick: we cast our `int` values to `void*` and vice versa, and store the value directly as the pointer value.

Here is a simple implementation for the `append()` method:

```
cdef append(self, int value):
    cqueue.queue_push_tail(self._c_queue, <void*>value)
```

Again, the same error handling considerations as for the `__cinit__()` method apply, so that we end up with this implementation:

```
cdef append(self, int value):
    if not cqueue.queue_push_tail(self._c_queue,
                                  <void*>value):
        python_exc.PyErr_NoMemory()
```

Adding an `extend()` method should now be straight forward:

```
cdef extend(self, int* values, Py_ssize_t count):
    """Append all ints to the queue.
    """
    cdef Py_ssize_t i
    for i in range(count):
        if not cqueue.queue_push_tail(
                self._c_queue, <void*>values[i]):
            python_exc.PyErr_NoMemory()
```

This becomes handy when reading values from a NumPy array, for example.

So far, we can only add data to the queue. The next step is to write the two methods to get the first element: `peek()` and `pop()`, which provide read-only and destructive read access respectively:

```
cdef int peek(self):
    return <int>cqueue.queue_peek_head(self._c_queue)

cdef int pop(self):
    return <int>cqueue.queue_pop_head(self._c_queue)
```

Simple enough. Now, what happens when the queue is empty? According to the documentation, the functions return a `NULL` pointer, which is typically not a valid value. Since we are simply casting to and from ints, we cannot distinguish anymore if the return value was `NULL` because the queue was empty or because the value stored in the queue was `0`. However, in Cython code, we would expect the first case to raise an exception, whereas the second case should simply return `0`. To deal with this, we need to special case this value, and check if the queue really is empty or not:

```
cdef int peek(self) except? 0:
    cdef int value = \
      <int>cqueue.queue_peek_head(self._c_queue)
    if value == 0:
        # this may mean that the queue is empty, or
        # that it happens to contain a 0 value
        if cqueue.queue_is_empty(self._c_queue):
            raise IndexError("Queue is empty")
    return value
```

The `except? 0` declaration is worth explaining. If the function was a Python function returning a Python object value, CPython would simply return `NULL` instead of a Python object to indicate a raised exception, which would immediately be propagated by the surrounding code. The problem is that any `int` value is a valid queue item value, so there is no way to explicitly indicate an error to the calling code.

The only way CPython (and Cython) can deal with this situation is to call `PyErr_Occurred()` when returning from a function to check if an exception was raised, and if so, propagate the exception. This obviously has a performance penalty. Cython therefore allows you to indicate which value is explicitly returned in the case of an exception, so that the surrounding code only needs to check for an exception when receiving this special value. All other values will be accepted almost without a penalty.

Now that the `peek()` method is implemented, the `pop()` method is almost identical. It only calls a different C function:

```
cdef int pop(self) except? 0:
    cdef int value = \
        <int>cqueue.queue_pop_head(self._c_queue)
    if value == 0:
        # this may mean that the queue is empty, or
        # that it happens to contain a 0 value
        if cqueue.queue_is_empty(self._c_queue):
            raise IndexError("Queue is empty")
    return value
```

Lastly, we can provide the Queue with an emptiness indicator in the normal Python way:

```
def __nonzero__(self):
    return not cqueue.queue_is_empty(self._c_queue)
```

Note that this method returns either `True` or `False` as the return value of the `queue_is_empty` function is declared as a `bint`.

Now that the implementation is complete, you may want to write some tests for it to make sure it works correctly. Especially doctests are very nice for this purpose, as they provide some documentation at the same time. To enable doctests, however, you need a Python API that you can call. C methods are not visible from Python code, and thus not callable from doctests.

A quick way to provide a Python API for the class is to change the methods from `cdef` to `cpdef`. This will let Cython generate two entry points, one that is callable from normal Python code using the Python call semantics and Python objects as arguments, and one that is callable from C code with fast C semantics and without requiring intermediate argument conversion from or to Python types.

The following listing shows the complete implementation that uses `cpdef` methods where possible. This feature is obviously not available for the `extend()` method, as the method signature is incompatible with Python argument types.

```
cimport cqueue
cimport python_exc

cdef class Queue:
    cdef cqueue.Queue* _c_queue
    def __cinit__(self):
        self._c_queue = cqueue.queue_new()
        if self._c_queue is NULL:
            python_exc.PyErr_NoMemory()

    def __dealloc__(self):
        if self._c_queue is not NULL:
            cqueue.queue_free(self._c_queue)

    cpdef append(self, int value):
        if not cqueue.queue_push_tail(self._c_queue,
                                      <void*>value):
            python_exc.PyErr_NoMemory()

    cdef extend(self, int* values, Py_ssize_t count):
        cdef Py_ssize_t i
        for i in range(count):
            if not cqueue.queue_push_tail(
                    self._c_queue, <void*>values[i]):
                python_exc.PyErr_NoMemory()

    cpdef int peek(self) except? 0:
        cdef int value = \
            <int>cqueue.queue_peek_head(self._c_queue)
        if value == 0:
            # this may mean that the queue is empty,
            # or that it happens to contain a 0 value
            if cqueue.queue_is_empty(self._c_queue):
                raise IndexError("Queue is empty")
        return value

    cpdef int pop(self) except? 0:
        cdef int value = \
            <int>cqueue.queue_pop_head(self._c_queue)
        if value == 0:
            # this may mean that the queue is empty,
            # or that it happens to contain a 0 value
            if cqueue.queue_is_empty(self._c_queue):
                raise IndexError("Queue is empty")
        return value

    def __nonzero__(self):
        return not cqueue.queue_is_empty(self._c_queue)
```

As a quick test with numbers from 0 to 9999 indicates, using this Queue from Cython code with C `int` values is about five times as fast as using it from Cython code with Python values, almost eight times faster than using it from Python code in a Python loop, and still more than twice as fast as using Python's highly optimised `collections.deque` type from Cython code with Python integers.

Unicode and passing strings

Similar to the string semantics in Python 3, Cython also strictly separates byte strings and unicode strings. Above all, this means that there is no automatic conversion between byte strings and unicode strings (except for what Python 2 does in string operations). All encoding and decoding must pass through an explicit encoding/decoding step.

It is, however, very easy to pass byte strings between C code and Python. When receiving a byte string from a C library, you can let Cython convert it into a Python byte string by simply assigning it to a Python variable:

```
cdef char* c_string = c_call_returning_a_c_string()
py_string = c_string
```

This creates a Python byte string object that holds a copy of the original C string. It can be safely passed around in Python code, and will be garbage collected when the last reference to it goes out of scope.

To convert the byte string back into a C `char*`, use the opposite assignment:

```
cdef char* other_c_string = py_string
```

This is a very fast operation after which `other_c_string` points to the byte string buffer of the Python string itself. It is tied to the life time of the Python string. When the Python string is garbage collected, the pointer becomes invalid. It is therefore important to keep a reference to the Python string as long as the `char*` is in use. Often enough, this only spans the call to a C function that receives the pointer as parameter. Special care must be taken, however, when the C function stores the pointer for later use. Apart from keeping a Python reference to the string, no manual memory management is required.

The above way of passing and receiving C strings is as simple that, as long as we only deal with binary data in the strings. When we deal with encoded text, however, it is best practice to decode the C byte strings to Python Unicode strings on reception, and to encode Python Unicode strings to C byte strings on the way out.

With a Python byte string object, you would normally just call the `.decode()` method to decode it into a Unicode string:

```
ustring = byte_string.decode('UTF-8')
```

You can do the same in Cython for a C string, but the generated code is rather inefficient for small strings. While Cython could potentially call the Python C-API function for decoding a C string from UTF-8 to Unicode (`PyUnicode_DecodeUTF8()`), the problem is that this requires passing the length of the C string, which Cython cannot know at compile time nor runtime. So it would have to call `strlen()` first, although the user code will already know the length of the string in almost all cases. Also, the encoded byte string might actually contain null bytes, so this isn't even a safe solution. It is therefore currently recommended to call the API functions directly:

```
# .pxd file that comes with Cython
cimport python_unicode

cdef char* c_string = NULL
cdef Py_ssize_t length = 0

# get pointer and length from a C function
get_a_c_string(&c_string, &length)

# decode the string to Unicode
ustring = python_unicode.PyUnicode_DecodeUTF8(
    c_string, length, 'strict')
```

It is common practice to wrap this in a dedicated function, as this needs to be done whenever receiving text from C. This could look as follows:

```
cimport python_unicode
cimport stdlib
cdef extern from "string.h":
    size_t strlen(char *s)

cdef unicode tounicode(char* s):
    return python_unicode.PyUnicode_DecodeUTF8(
        s, strlen(s), 'strict')

cdef unicode tounicode_with_length(
        char* s, size_t length):
    return python_unicode.PyUnicode_DecodeUTF8(
        s, length, 'strict')

cdef unicode tounicode_with_length_and_free(
        char* s, size_t length):
    try:
        return python_unicode.PyUnicode_DecodeUTF8(
            s, length, 'strict')
    finally:
        stdlib.free(s)
```

Most likely, you will prefer shorter function names in your code based on the kind of string being handled. Different types of content often imply different ways of handling them on reception. To make the code more readable and to anticipate future changes, it is good practice to use separate conversion functions for different types of strings.

The reverse way, converting a Python unicode string to a C `char*`, is pretty efficient by itself, assuming that what you actually want is a memory managed byte string:

```
py_byte_string = py_unicode_string.encode('UTF-8')
cdef char* c_string = py_byte_string
```

As noted above, this takes the pointer to the byte buffer of the Python byte string. Trying to do the same without keeping a reference to the intermediate byte string will fail with a compile error:

```
# this will not compile !
cdef char* c_string = py_unicode_string.encode('UTF-8')
```

Here, the Cython compiler notices that the code takes a pointer to a temporary string result that will be garbage collected after the assignment. Later access to the invalidated pointer will most likely result in a crash. Cython will therefore refuse to compile this code.

Caveats

Since Cython mixes C and Python semantics, some things may be a bit surprising or unintuitive. Work always goes on to make Cython more natural for Python users, so this list may change in the future.

- `10**-2 == 0`, instead of `0.01` like in Python.
- Given two typed `int` variables `a` and `b`, `a % b` has the same sign as the first argument (following C semantics) rather then having the same sign as the second (as in Python). This will change in Cython 0.12.
- Care is needed with unsigned types. `cdef unsigned n = 10; print(range(-n, n))` will print an empty list, since `-n` wraps around to a large positive integer prior to being passed to the `range` function.
- Python's `float` type actually wraps C `double` values, and Python's `int` type wraps C `long` values.

Further reading

The main documentation is located at http://docs.cython.org/. Some recent features might not have documentation written yet, in such cases some notes can usually be found in the form of a Cython Enhancement Proposal (CEP) on http://wiki.cython.org/enhancements.

[Seljebotn09] contains more information about Cython and NumPy arrays. If you intend to use Cython code in a multi-threaded setting, it is essential to read up on Cython's features for managing the Global Interpreter Lock (the GIL). The same paper contains an explanation of the GIL, and the main documentation explains the Cython features for managing it.

Finally, don't hesitate to ask questions (or post reports on successes!) on the Cython users mailing list [UserList]. The Cython developer mailing list, [DevList], is also open to everybody. Feel free to use it to report a bug, ask for guidance, if you have time to spare to develop Cython, or if you have suggestions for future development.

Related work

Pyrex [Pyrex] is the compiler project that Cython was originally based on. Many features and the major design decisions of the Cython language were developed by Greg Ewing as part of that project. Today, Cython supersedes the capabilities of Pyrex by providing a higher compatibility with Python code and Python semantics, as well as superior optimisations and better integration with scientific Python extensions like NumPy.

ctypes [ctypes] is a foreign function interface (FFI) for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python code. Compared to Cython, it has the major advantage of being in the standard library and being usable directly from Python code, without any additional dependencies. The major drawback is its performance, which suffers from the Python call overhead as all operations must pass through Python code first. Cython, being a compiled language, can avoid much of this overhead by moving more functionality and long-running loops into fast C code.

SWIG [SWIG] is a wrapper code generator. It makes it very easy to parse large API definitions in C/C++ header files, and to generate straight forward wrapper

code for a large set of programming languages. As opposed to Cython, however, it is not a programming language itself. Thin wrappers are easy to generate, but the more functionality a wrapper needs to provide, the harder it gets to implement it with SWIG. Cython, on the other hand, makes it very easy to write very elaborate wrapper code specifically for the Python language. Also, there exists third party code for parsing C header files and using it to generate Cython definitions and module skeletons.

ShedSkin [ShedSkin] is an experimental Python-to-C++ compiler. It uses profiling information and very powerful type inference engine to generate a C++ program from (restricted) Python source code. The main drawback is has no support for calling the Python/C API for operations it does not support natively, and supports very few of the standard Python modules.

Appendix: Installing MinGW on Windows

1. Download the MinGW installer from http://mingw.org. (As of this writing, the download link is a bit difficult to find; it's under "About" in the menu on the left-hand side). You want the file entitled "Automated MinGW Installer" (currently version 5.1.4).
2. Run it and install MinGW. Only the basic package is strictly needed for Cython, although you might want to grab at least the C++ compiler as well.
3. You need to set up Windows' "PATH" environment variable so that includes e.g. "c:\mingw\bin" (if you installed MinGW to "c:\mingw"). The following web-page describes the procedure in Windows XP (the Vista procedure is similar): http://support.microsoft.com/kb/310519
4. Finally, tell Python to use MinGW as the default compiler (otherwise it will try for Visual C). If Python is installed to "c:\Python26", create a file named "c:\Python26\Lib\distutils\distutils.cfg" containing:

```
[build]
compiler = mingw32
```

The [WinInst] wiki page contains updated information about this procedure. Any contributions towards making the Windows install process smoother is welcomed; it is an unfortunate fact that none of the regular Cython developers have convenient access to Windows.

References

[Cython] G. Ewing, R. W. Bradshaw, S. Behnel, D. S. Seljebotn et al., The Cython compiler, http://cython.org.

[Python] G. van Rossum et al., The Python programming language, http://python.org.

[Sage] W. Stein et al., Sage Mathematics Software, http://sagemath.org

[EPD] Enthought, Inc., *The Enthought Python Distribution* http://www.enthought.com/products/epd.php

[Pythonxy] P. Raybault, http://www.pythonxy.com/

[Jython] J. Huginin, B. Warsaw, F. Bock, et al., Jython: Python for the Java platform, http://www.jython.org/

[Seljebotn09] D. S. Seljebotn, Fast numerical computations with Cython, Proceedings of the 8th Python in Science Conference, 2009.

[NumPy] T. Oliphant et al., NumPy, http://numpy.scipy.org/

[CAlg] S. Howard, C Algorithms library, http://c-algorithms.sourceforge.net/

[Pyrex] G. Ewing, Pyrex: C-Extensions for Python, http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/

[ShedSkin] M. Dufour, J. Coughlan, ShedSkin, http://code.google.com/p/shedskin/

[PyPy] The PyPy Group, PyPy: a Python implementation written in Python, http://codespeak.net/pypy.

[IronPython] J. Hugunin et al., http://www.codeplex.com/IronPython.

[SWIG] D.M. Beazley et al., SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++, http://www.swig.org.

[WinInst] http://wiki.cython.org/InstallingOnWindows

[ctypes] T. Heller et al., http://docs.python.org/library/ctypes.html.

[UserList] Cython users mailing list: http://groups.google.com/group/cython-users

[DevList] Cython developer mailing list: http://codespeak.net/mailman/listinfo/cython-dev.

Fast numerical computations with Cython

Dag Sverre Seljebotn (dagss@student.matnat.uio.no) – *University of Oslo*[abc], NORWAY

[a]Institute of Theoretical Astrophysics, University of Oslo, P.O. Box 1029 Blindern, N-0315 Oslo, Norway
[b]Department of Mathematics, University of Oslo, P.O. Box 1053 Blindern, N-0316 Oslo, Norway
[c]Centre of Mathematics for Applications, University of Oslo, P.O. Box 1053 Blindern, N-0316 Oslo, Norway

Cython has recently gained popularity as a tool for conveniently performing numerical computations in the Python environment, as well as mixing efficient calls to natively compiled libraries with Python code. We discuss Cython's features for fast NumPy array access in detail through examples and benchmarks. Using Cython to call natively compiled scientific libraries as well as using Cython in parallel computations is also given consideration. We conclude with a note on possible directions for future Cython development.

Introduction

Python has in many fields become a popular choice for scientific computation and visualization. Being designed as a general purpose scripting language without a specific target audience in mind, it tends to scale well as simple experiments grow to complex applications. From a numerical perspective, Python and associated libraries can be regarded mainly as a convenient shell around computational cores written in natively compiled languages, such as C, C++ and Fortran. For instance, the Python-specific SciPy [SciPy] library contains over 200 000 lines of C++, 60 000 lines of C, and 75 000 lines of Fortran, compared to about 70 000 lines of Python code.

There are several good reasons for such a workflow. First, if the underlying compiled library is usable in its own right, and also has end-users writing code in MATLAB, C++ or Fortran, it may make little sense to tie it too strongly to the Python environment. In such cases, writing the computational cores in a compiled language and using a Python wrapper to direct the computations can be the ideal workflow. Secondly, as we will see, the Python interpreter is too slow to be usable for writing low-level numerical loops. This is particularly a problem for computations which can not be expressed as operations on entire arrays.

Cython is a programming language based on Python, with additional syntax for optional static type declarations. The Cython compiler is able to translate Cython code into C code making use of the CPython C API [CPyAPI], which can in turn be compiled into a module loadable into any CPython session. The end-result can perhaps be described as a language which allows one to use Python and C interchangeably in the same code. This has two important applications. First, it is useful for creating Python wrappers around natively compiled code, in particular in situations where one does not want a 1:1 mapping between the library API and the Python API, but rather a higher-level Pythonic wrapper. Secondly, it allows incrementally speeding up Python code. One can start out with a simple Python prototype, then proceed to incrementally add type information and C-level optimization strategies in the few locations that really matter. While being a superset of Python is a goal for Cython, there is currently a few incompatibilities and unsupported constructs. The most important of these is inner functions and generators (closure support).

In this paper we will discuss Cython from a numerical computation perspective. Code is provided for illustration purposes and the syntax is not explained in full, for a detailed introduction to Cython we refer to [Tutorial] and [Docs]. [Wilbers] compares Cython with similar tools ([f2py], [Weave], [Instant] and [Psyco]). The comparison is for speeding up a particular numerical loop, and both speed and usability is discussed. Cython here achieves a running time 1.6 times that of the Fortran implementation. We note that had the arrays been declared as contiguous at compile-time, this would have been reduced to 1.3 times the time of Fortran. [Ramach] is a similar set of benchmarks, which compare Pyrex and other tools with a pure Python/NumPy implementation. Cython is based on [Pyrex] and the same results should apply, the main difference being that Cython has friendlier syntax for accessing NumPy arrays efficiently.

Fast array access

Fast array access, added to the Cython language by D. S. Seljebotn and R. W. Bradshaw in 2008, was an important improvement in convenience for numerical users. The work is based on PEP 3118, which defines a C API for direct access to the array data of Python objects acting as array data containers[1]. Cython is able to treat most of the NumPy array data types as corresponding native C types. Since Cython 0.11.2, complex floating point types are supported, either through the C99 complex types or through Cython's own implementation. Record arrays are mapped to arrays of C structs for efficient access. Some data types are not supported, such as string/unicode arrays, arrays with non-native endianness and boolean arrays. The latter can however be treated as 8-bit integer arrays in Cython. [Tutorial] contains further details.

[1]PEP 3118 is only available on Python 2.6 and greater, therefore a backwards-compatibility mechanism is also provided to emulate the protocol on older Python versions. This mechanism is also used in the case of NumPy arrays, which do not yet

To discuss this feature we will start with the example of naive matrix multiplication. Beginning with a pure Python implementation, we will incrementally add optimizations. The benchmarks should help Cython users decide how far one wants to go in other cases. For $C = AB$ the computation is

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

where n is the number of columns in A and rows in B. A basic implementation in pure Python looks like this:

```
def matmul(A, B, out):
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            s = 0
            for k in range(A.shape[1]):
                s += A[i, k] * B[k, j]
            out[i,j] = s
```

For clarity of exposition, this skips the details of sanity checking the arguments. In a real setting one should probably also automatically allocate `out` if not provided by the caller.

Simply compiling this in Cython results in a about 1.15x speedup over Python. This minor speedup is due to the compiled C code being faster than Python's byte code interpreter. The generated C code still uses the Python C API, so that e.g. the array lookup `A[i, k]` translates into C code very similar to:

```
tmp = PyTuple_New(2);
if (!tmp) { err_lineno = 21; goto error; }
Py_INCREF(i);
PyTuple_SET_ITEM(tmp, 0, i);
Py_INCREF(k);
PyTuple_SET_ITEM(tmp, 1, k);
A_ik = PyObject_GetItem(A, tmp);
if (!A_ik) { err_lineno = 21; goto error; }
Py_DECREF(tmp);
```

The result is a pointer to a Python object, which is further processed with `PyNumber_Multiply` and so on. To get any real speedup, types must be added:

```
import numpy as np
cimport numpy as np
ctypedef np.float64_t dtype_t
def matmul(np.ndarray[dtype_t, ndim=2] A,
           np.ndarray[dtype_t, ndim=2] B,
           np.ndarray[dtype_t, ndim=2] out=None):
  cdef Py_ssize_t i, j, k
  cdef dtype_t s
  if A is None or B is None:
      raise ValueError("Input matrix cannot be None")
  for i in range(A.shape[0]):
      for j in range(B.shape[1]):
          s = 0
          for k in range(A.shape[1]):
              s += A[i, k] * B[k, j]
          out[i,j] = s
```

In our benchmarks this results in a speedup of between 180-190 times over pure Python. The exact factor varies depending on the size of the matrices. If they are not small enough to be kept in the CPU cache, the data must be transported repeatedly over the memory bus. This is close to equally expensive for Python and Cython and thus tends to slightly diminish any other effects. Table 1 has the complete benchmarks, with one in-cache benchmark and one out-of-cache benchmark in every case.

Note however that the speedup does not come without some costs. First, the routine is now only usable for 64-bit floating point. Arrays containing any other data type will result in a `ValueError` being raised. Second, it is necessary to ensure that typed variables containing Python objects are not `None`. Failing to do so can result in a crash or data corruption if `None` is passed to the routine.

The generated C source for the array lookup `A[i, k]` now looks like this:

```
tmp_i = i; tmp_k = k;
if (tmp_i < 0) tmp_i += A_shape_0;
if (tmp_i < 0 || tmp_i >= A_shape_1) {
  PyErr_Format(<...>);
  err_lineno = 33; goto error;
}
if (tmp_k < 0) tmp_k += A_shape_1;
if (tmp_k < 0 || tmp_k >= A_shape_1) {
  PyErr_Format(<...>);
  err_lineno = 33; goto error;
}
A_ik = *(dtype_t*)(A_data +
  tmp_i * A_stride_0 + tmp_k * A_stride_1);
```

This is a lot faster because there are no API calls in a normal situation, and access of the data happens directly to the underlying memory location. The initial conditional tests are there for two reasons. First, an if-test is needed to support negative indices. With the usual Python semantics, `A[-1, -1]` should refer to the lower-right corner of the matrix. Second, it is necessary to raise an exception if an index is out of bounds.

Such if-tests can bring a large speed penalty, especially in the middle of the computational loop. It is therefore possible to instruct Cython to turn off these features through compiler directives. The following code disables support for negative indices (`wraparound`) and bounds checking:

```
cimport cython
@cython.boundscheck(False)
@cython.wraparound(False)
def matmul(np.ndarray[dtype_t, ndim=2] A,
           np.ndarray[dtype_t, ndim=2] B,
           np.ndarray[dtype_t, ndim=2] out=None):
    <...>
```

This removes all the if-tests from the generated code. The resulting benchmarks indicate around 800 times speedup at this point in the in-cache situation, 700 times out-of-cache. Only disabling one of either `wraparound` or `boundscheck` will not have a significant impact because there will still be if-tests left.

One trade-off is that should one access the arrays out of bounds, one will have data corruption or a program crash. The normal procedure is to leave bounds checking on until one is completely sure that the code is correct, then turn it off. In this case we are not using negative indices, and it is an easy decision to turn them off. Even if we were, it would be faster to manually add code to calculate the corresponding positive

support PEP 3118, even on Python 2.6.

index. `wraparound` is mainly enabled by default to reduce the number of surprises for casual Cython users, and one should rarely leave it on.
In addition to per-function level like here, compiler directives can also be specified globally for the source file or for individual code blocks. See [directives] for further information.

	80x80	1500x1500
	Units: MFLOPS	
Optimal layout		
Python	0.94	0.98
Cython	1.08	1.12
Added types	179	177
`boundscheck/wraparound`	770	692
`mode="c"/mode="fortran"`	981	787
BLAS `ddot` (ATLAS)	1282	911
Intel C	2560	1022
gfortran A^TB	1113	854
Intel Fortran A^TB	2833	1023
NumPy `dot`	3656	4757
Worst-case layout		
Python	0.94	0.97
`boundscheck/wraparound`	847	175
BLAS `ddot` (ATLAS)	910	183
gfortran AB^T	861	94
Intel Fortran AB^T	731	94

***Table 1**: Matrix multiplication benchmarks on an Intel Xeon 3.2 GHz, 2 MB cache, SSE2. The smaller data set fits in cache, while the larger does not. Keep in mind that different implementations have different constant-time overhead, which e.g. explains that NumPy `dot` does better for larger dataset.*

Caring about memory layout

Both C/C++ and Fortran assume that arrays are stored as one contiguous chunk in memory. NumPy arrays depart from this tradition and allows for arbitrarily strided arrays. Consider the example of `B = A[::-2,:]`. That is, let B be the array consisting of every other row in A, in reverse order. In many environments, the need for representing B contiguously in memory mandates that a copy is made in such situations. NumPy supports a wider range of array memory layouts and can in this situation construct B as a new view to the same data that A refers to. The benefit of the NumPy approach is that it is more flexible, and allows avoiding copying of data. This is especially important if one has huge data sets where the main memory might only be able to hold one copy at the time. With choice does however come responsibility. In order to gain top performance with numerical computations it is in general crucial to pay attention to memory layout.

In the example of matrix multiplication, the first matrix is accessed row-by-row and the second column-by-column. The first matrix should thus be stored with contiguous rows, "C contiguous", while the second should be stored with contiguous columns, "Fortran contiguous". This will keep the distance between subsequent memory accesses as small as possible. In an out-of-cache situation, our fastest matmul routine so far does around 700 times better than pure Python when presented with matrices with optimal layout, but only around 180 times better with the worst-case layout. See table 1.
The second issue concerning memory layout is that a price is paid for the generality of the NumPy approach: In order to address arbitrarily strided arrays, an extra integer multiplication operation must be done per access. In the `matmul` implementation above, `A[i,k]` was translated to the following C code:

```
A_ik = *(dtype_t*)(A_data + i * A_stride_0
                          + j * A_stride_1);
```

By telling Cython at compile-time that the arrays are contiguous, it is possible to drop the innermost stride multiplication. This is done by using the "mode" argument to the array type:

```
def matmul(
  np.ndarray[dtype_t, ndim=2, mode="c"] A,
  np.ndarray[dtype_t, ndim=2, mode="fortran"] B,
  np.ndarray[dtype_t, ndim=2] out=None):
    <...>
```

The in-cache benchmark now indicate a 780x speedup over pure Python. The out-of-cache improvement is smaller but still noticeable. Note that no restrictions is put on the `out` argument. Doing so did not lead to any significant speedup as `out` is not accessed in the inner loop.
The catch is, of course, that the routine will now reject arrays which does not satisfy the requirements. This happens by raising a ValueError. One can make sure that arrays are allocated using the right layout by passing the "order" argument to most NumPy array constructor functions, as well as the copy() method of NumPy arrays. Furthermore, if an array A is C-contiguous, then the transpose, A.T, will be a Fortran-contiguous view of the same data.
The number of memory accesses of multiplying two $n \times n$ matrices scale as $O(n^3)$, while copying the matrices scale as $O(n^2)$. One would therefore expect, given that enough memory is available, that making a temporary copy pays off once n passes a certain threshold. In this case benchmarks indicate that the threshold is indeed very low (in the range of $n = 10$) and one would typically copy in all situations. The NumPy functions `ascontiguousarray` and `asfortranarray` are helpful in such situations.

Calling an external library

The real world usecase for Cython is to speed up custom numerical loops for which there are no prior implementations available. For a simple example like matrix

multiplication, going with existing implementations is always better. For instance, NumPy's `dot` function is about 6 times faster than our fastest Cython implementation, since it uses smarter algorithms. Under the hood, `dot` makes a call to the `dgemm` function in the Basic Linear Algebra Software API ([BLAS], [ATLAS])[2].

One advantage of Cython is how easy it is to call native code. Indeed, for many, this is the entire point of using Cython. We will demonstrate these features by calling BLAS for the inner products only, rather than for the whole matrix multiplication. This allows us to stick with the naive matrix multiplication algorithm, and also demonstrates how to mix Cython code and the use of external libraries. The BLAS API must first be declared to Cython:

```
cdef extern from "cblas.h":
    double ddot "cblas_ddot"(int N,
                             double *X, int incX,
                             double *Y, int incY)
```

The need to re-declare functions which are already declared in C is unfortunate and an area of possible improvement for Cython. Only the declarations that are actually used needs to be declared. Note also the use of C pointers to represent arrays. BLAS also accepts strided arrays and expects the strides passed in the `incX` and `incY` arguments. Other C APIs will often require a contiguous array where the stride is fixed to one array element; in the previous section it was discussed how one can ensure that arrays are contiguous. The matrix multiplication can now be performed like this:

```
ctypedef np.float64_t dtype_t
def matmul(np.ndarray[dtype_t, ndim=2] A,
           np.ndarray[dtype_t, ndim=2] B,
           np.ndarray[dtype_t, ndim=2] out):
  cdef Py_ssize_t i, j
  cdef np.ndarray[dtype_t, ndim=1] A_row, B_col
  for i in range(A.shape[0]):
      A_row = A[i,:]
      for j in range(B.shape[1]):
          B_col = B[:, j]
          out[i,j] = ddot(
            A_row.shape[0],
            <dtype_t*>A_row.data,
            A_row.strides[0] // sizeof(dtype_t),
            <dtype_t*>B_col.data,
            B_col.strides[0] // sizeof(dtype_t))
```

This demonstrates how NumPy array data can be passed to C code. Note that NumPy strides are in number of bytes, while BLAS expects them in number of elements. Also, because the array variables are typed, the "data" attributes are C pointers rather than Python buffer objects.

Unfortunately, this results in a slowdown for moderate n. This is due to the slice operations. Operations like `A_row = A[i,:]` is a Python operation and results in Python call overhead and the construction of several new objects. Cython is unlikely to ever optimize slicing of np.ndarray variables because it should remain possible to use subclasses of ndarray with a different slicing behaviour[3]. The new memory view type, discussed below, represents a future solution to this problem. Another solution is to do the slice calculations manually and use C pointer arithmetic:

```
out[i,j] = ddot(
    A.shape[1],
    <dtype_t*>(A.data + i*A.strides[0]),
    A.strides[1] // sizeof(dtype_t),
    <dtype_t*>(B.data + j*B.strides[1]),
    B.strides[0] // sizeof(dtype_t))
```

This version leads to over 1300 times speedup over pure Python in the optimal, in-cache situation. This is due to BLAS using the SSE2 instruction set, which enables doing two double-precision floating point multiplications in one CPU instruction. When non-contiguous arrays are used the performance drops to 970 times that of pure Python (in-cache) as SSE2 can no longer be used.

For more advanced array data passing, Cython makes it easy to make use of NumPy's C API. Consider for instance a custom C library which returns a pointer to some statically allocated data, which one would like to view as a NumPy array. Using Cython and the NumPy C API this is easily achieved:

```
cimport numpy as np
cdef extern from "mylib.h":
    cdef int get_my_data(double** out_data,
                         int* out_size)
def my_data_as_ndarray():
    cdef np.npy_intp* shape = [0]
    cdef int arr_length
    cdef double* arr_ptr
    if get_my_data(&arr_ptr, &arr_length) != 0:
        raise RuntimeError("get_my_data failed")
    shape[0] = arr_length
    return np.PyArray_SimpleNewFromData(1, shape,
      np.NPY_DOUBLE, <void*>arr_ptr)
```

SSE and vectorizing C compilers

What about using SSE directly in a Cython program? One possibility is to use the SSE API of the C compiler. The details varies according to the C compiler used, but most C compilers offers a set of special functions which corresponds directly to SSE instructions. These can be used as any other C function, also from Cython code. In general, such code tends to be somewhat more complicated, as the first element in every loop must be treated as a special case (in case the elements involved are not aligned on 128-bit boundaries in memory, as required by SSE). We have not included code or benchmarks for this approach.

Another popular approach to SSE is to use a "vectorizing" C compiler. For instance both Intel's C compiler, [ICC], and the GNU C compiler, [GCC], can recognize

[2]BLAS is an API with many implementations; the benchmarks in this paper is based on using the open-source ATLAS implementation, custom-compiled on the host by Sage [Sage].

[3]This principle has of course already been violated, as one could change the behaviour of the element indexing in a subclass as well. The policy is however not to go further in this direction. Note also that only indexing with typed integer variables is optimized; A[i, some_untyped_var] is not optimized as the latter index could e.g. point to a Python slice object.

certain loops as being fit for SSE optimization (and other related optimizations). Note that this is a non-trivial task as multiple loop iterations are combined, and the loop must typically be studied as a whole. Unfortunately, neither ICC v. 11 nor GCC v. 4.3.3 managed to vectorize the kind of code Cython outputs by default. After some manual code massaging we managed to have ICC compile a vectorized version which is included in the benchmarks. We did not manage to get GCC to vectorize the kind of sum-reduce loop used above.

It appears that Cython has some way to go to be able to benefit from vectorizing C compilers. Improving Cython so that the generated C code is more easily vectorizable should be possible, but has not been attempted thus far. Another related area of possible improvement is to support generating code containing the C99 restrict modifier, which can be used to provide guarantees that arrays do not overlap. GCC (but not ICC) needs this to be present to be able to perform vectorization.

Linear time: Comparisons with NumPy

We turn to some examples with linear running time. In all cases the computation can easily be expressed in terms of operations on whole arrays, allowing comparison with NumPy.

First, we consider finding elements with a given value in a one-dimensional array. This operation can be performed in NumPy as:

```
haystack = get_array_data()
result = np.nonzero(haystack == 20)
```

This results in a array of indices, listing every element equal to 20. If the goal is simply to extract the first such element, one can instead use a very simple loop in Cython. This avoids constructing a temporary array and the result array. A simple Cython loop then performed about five times faster than the NumPy code in our benchmarks. The point here is merely that Cython allows easily writing code specifically tailored for the situation at hand, which sometimes can bring speed benefits.

Another example is that of operating on a set of array elements matching some filter. For instance, consider transforming all 2D points within a given distance from zero:

```
Point_dtype = np.dtype([('x', np.float64),
                        ('y', np.float64)])
points = load_point_data(filename, Point_dtype)
radius = 1.2
tmp = points['x']**2
tmp += points['y']**2
pointset = tmp < radius**2
points['x'][pointset] *= 0.5
points['y'][pointset] *= 0.3
```

This code uses a number of temporary arrays to perform the calculation. In an in-cache situation, the overhead of constructing the temporary Python arrays becomes noticeable. In an out-of-cache situation, the data has to be transported many times over the memory bus. The situation is worsened by using a record array (as more data is transported over the bus in total, and less cache becomes available). Using separate arrays for `x` and `y` results in a small speedup; both array layouts are included in the benchmarks.

A Cython loop is able to do the operation in a single pass, so that the data is only transported once:

```
cdef packed struct Point:
    np.float64_t x, y

def transform_within_circle(np.ndarray[Point] points,
                            np.float64_t radius):
  cdef Py_ssize_t i
  cdef Point p
  cdef np.float64_t radius_sq = radius**2
  for i in range(points.shape[0]):
      p = points[i]
      if p.x**2 + p.y**2 < radius_sq:
          p.x *= 0.5
          p.y *= 0.3
          points[i] = p
```

This is 10 times faster than the NumPy code for a large data set, due to the heavily reduced memory bus traffic. NumPy also uses twice as much memory due to the temporaries. If the data set is several gigabytes, then the additional memory used by NumPy could mean the difference between swapping and not swapping to disk. For Cython, operating on separate `x` and `y` arrays is slightly slower. See table 2.

Finally, it would have been possible to separate the filter and the transform by passing a callback to be called in each iteration in the loop. By making use of Cython extension type classes, which have faster method dispatches than Python classes, the penalty of such an approach is only around 20-25%. [Tutorial] demonstrates such callbacks.

	Million elements processed per second 2×10^7 element test set	
	Records	Seperate
Python loop	0.028	0.069
NumPy	9.5	10
Cython plain	95	79
Cython optimized	110	100
Cython w/callback	79	73

***Table 2**: Benchmarks for operating on points within a circle. The optimized Cython version and the callback version both has `boundscheck`/`wraparound` turned off and `mode='c'` specified. All benchmarks on an Intel Xeon 3.2 GHz, 2 MB cache. All points were within the circle in the test data set.*

Parallel computation

When discussing parallel computation there is an important distinction between shared-memory models and message passing models. We start with discussing

the shared memory case. A common approach in parallel numerical code is to use OpenMP. While not difficult to support in principle, OpenMP is currently not available in Cython. Instead, Python threads must be used. This comes with some problems, but they can be worked around.

Threads is a problem with CPython because every operation involving a Python object must be done while holding the Global Interpreter Lock (GIL). The result is that pure Python scripts are typically unable to utilize more than one CPU core, even if many threads are used. It should be noted that for many computational scripts this does not matter. If the bulk of the computation happens in wrapped, native code (like in the case of NumPy or SciPy) then the GIL is typically released during the computation. For Cython the situation is worse. Once inside Cython code, the GIL is by default held until one returns to the Python caller. The effect is that threads doesn't switch at all. Whereas a pure Python script will tend to switch between threads on a single CPU core, a Cython program will by default tend to not switch threads at all.

The solution is to use Cython language constructs to manually release the GIL. One can then achieve proper multi-threading on many cores. The catch is that no Python operations are allowed when the GIL is released; for instance, all variables used must be typed with a C type. Optimized NumPy array lookups are allowed. The Cython compiler will help enforce these rules. Example:

```
@cython.boundscheck(False)
def find_first(np.ndarray[np.int64_t] haystack,
               np.int64_t needle):
    cdef Py_ssize_t i, ret = -1
    with nogil:
        for i from 0 <= i < haystack.shape[0]:
            if haystack[i] == needle:
                ret = i; break
    return ret
```

Without `nogil`, invocations from separate threads would be serialized. Returning the result is a Python operation, so that has to be put outside of the `nogil` block . Furthermore, `boundscheck` must be turned off as raising an `IndexError` would require the GIL[4]. [Docs] contains further information on the various primitives for managing the GIL (search for "nogil").

The message passing case is much simpler. Several Python interpreters are launched, each in its own process, so that the GIL is not an issue. A popular approach is mpi4py [mpi4py], together with an MPI implementation (such as OpenMPI [OpenMPI]). mpi4py very conveniently allows passing full Python objects between computational nodes through Python pickling[5]. It is also possible to efficiently pass NumPy arrays. mpi4py is itself written in Cython, and ships with the Cython compile-time definitions necessary to communicate directly with the underlying MPI C API. It is thus possible to use both interchangeably:

```
from mpi4py import MPI
from mpi4py cimport MPI
from mpi4py cimport mpi_c

cdef MPI.Comm comm = MPI.COMM_WORLD

if comm.Get_rank() == 0:
    # High-level send of Python object
    comm.send({'a': any_python_object, 'b': other},
              to=1)
    for <...a lot of small C-typed messages...>:
        # Fast, low-level send of typed C data
        mpi_c.MPI_Send(<...>, comm.ob_mpi)
elif ...
```

This is useful e.g. in situations where one wants to pass typed Cython variables and does not want to bother with conversion back and forth to Python objects. This also avoids the overhead of the Python call to mpi4py (although in practice there is likely to be other, larger bottlenecks). While contrived, this example should demonstrate some of the flexibility of Cython with regards to native libraries. mpi4py can be used for sending higher-level data or a few big messages, while the C API can be used for sending C data or many small messages.

The same principle applies to multi-threaded code: It is possible, with some care, to start the threads through the Python API and then switch to e.g. native OS thread mutexes where any Python overhead would become too large. The resulting code would however be platform-specific as Windows and Unix-based systems have separate threading APIs.

Conclusions and future work

While the examples shown have been simple and in part contrived, they explore fundamental properties of loops and arrays that should apply in almost any real-world computation. For computations that can only be expressed as for-loops, and which is not available in a standard library, Cython should be a strong candidate. Certainly, for anything but very small amounts of data, a Python loop is unviable. The choice stands between Cython and other natively compiled technologies. Cython may not automatically produce quite as optimized code as e.g. Fortran, but we believe it is fast enough to still be attractive in many cases because of the high similarity with Python. With Cython and NumPy, copying of non-contiguous arrays is always explicit, which can be a huge advantage compared to some other technologies (like Fortran) when dealing with very large data sets.

For the algorithms which are expressible as NumPy operations, the speedup is much lower, ranging from no speedup to around ten times. The Cython code is usually much more verbose and requires more decisions to be made at compile-time. Use of Cython in

[4] Finally, a different for loop syntax must be used, but this restriction will disappear in Cython 0.12.

[5] In addition to normal Python classes, Cython supports a type more efficient classes known as "extension types". For efficiency reasons these need explicit implementations provided for pickling and unpickling. See "Pickling and unpickling extension types" in [CPyPickle].

these situations seems much less clear cut. A good approach is to prototype using pure Python, and, if it is deemed too slow, optimize the important parts after benchmarks or code profiling.

Cython remains in active development. Because of the simple principles involved, new features are often easy to add, and are often the result of personal itch-scratching. Sometimes the experience has been that it is quicker to add a feature to Cython than to repeatedly write code to work around an issue. Some highlights of current development:

- Support for function-by-function profiling through the Python cProfile module was added in 0.11.3.
- Inner functions (closures) are maturing and will be released soon.
- Cython benefited from two Google Summer of Code [GSoC] projects over summer of 2009, which will result in better support for calling C++ and Fortran code.

One important and often requested feature for numerical users is template support. This would make it possible to make a single function support all array data types, without code duplication. Other possible features are improved parallel programming support, like OpenMP primitives. While no work is currently going on in these areas, the Cython developers remain conscious about these shortcomings.

One important future feature for numerical users is the new *memory view type*. K. W. Smith and D. S. Seljebotn started work on this in summer 2009 as part of Smith's Google Summer of Code.

The new Python buffer protocol based on PEP 3118 promise a shift in focus for array data. In Python 2.6 and greater, any Python object can export any array-like data to natively compiled code (like Cython code) in an efficient and standardized manner. In some respects this represents adoption of some NumPy functionality into the Python core. With this trend, it seems reasonable that Cython should provide good mechanisms for working with PEP 3118 buffers independently of NumPy. Incidentally, this will also provide a nice unified interface for interacting with C and Fortran arrays in various formats. Unlike NumPy, PEP 3118 buffers also supports pointer indirection-style arrays, sometimes used in C libraries.

With this new feature, the matrix multiplication routine could have been declared as:

```
def matmul(double[:,:] A,
           double[:,:] B,
           double[:,:] out):
    <...>
```

Using this syntax, a buffer is acquired from the arguments on entry. The interface of the argument variables are entirely decided by Cython, and it is not possible to use Python object operations. NumPy array methods like `A.mean()` will therefore no longer work. Instead, one will have to call `np.mean(A)` (which will work once NumPy supports PEP 3118). The advantage is that when Cython defines the interface, further optimizations can be introduced. Slices and arithmetic operations are not currently subject for optimization because of polymorphism. For instance, it would currently be impossible for Cython to optimize the multiplication operator, as it means different things for `ndarray` and its subclass `matrix`.

A nice benefit of the chosen syntax is that individual axis specifications becomes possible:

```
def matmul(double[:,::contig] A,
           double[::contig,:] B,
           double[:,:] out):
    <...>
```

Here, A is declared to have contiguous rows, without necessarily being contiguous as a whole. For instance, slicing along the first dimension (like `A[::3,:]`) would result in such an array. The `matmul` function can still benefit from the rows being declared as contiguous. Finally, we contemplate specific syntax for automatically making contiguous copies:

```
def matmul(in double[:,::contig] A,
           in double[::contig,:] B,
           double[:,:] out):
    <...>
```

This is particularly convenient when interfacing with C or Fortran libraries which demand such arrays.

Once basic memory access support is supported, it becomes possible to add optimizations for whole-array "vectorized" operations. For instance, this example could be supported:

```
cdef extern from "math.h":
    double sqrt(double)

def func(double[:] x, double[:] y):
    return sqrt(x**2 + y**2)
```

This would translate into a loop over the elements of `x` and `y`. Both a naive translation to C code, SSE code, GPU code, and use of BLAS or various C++ linear algebra libraries could eventually be supported. Whether Cython will actually move in this direction remains an open question. For now we simply note that even the most naive implementation of the above would lead to at least a 4 times speedup for large arrays over the corresponding NumPy expression; again due to NumPy's need for repeatedly transporting the data over the memory bus.

Acknowledgments

DSS wish to thank Stefan Behnel and Robert Bradshaw for enthusiastically sharing their Cython knowledge, and his advisor Hans Kristian Eriksen for helpful advice on writing this article. Cython is heavily based on [Pyrex] by Greg Ewing. The efficient NumPy array access was added to the language as the result of financial support by the Google Summer of Code program and Enthought Inc. The Centre of Mathematics for Applications at the University of Oslo and the Python Software Foundation provided funding to attend the SciPy '09 conference.

References

[Wilbers] I. M. Wilbers, H. P. Langtangen, Å. Ødegård, *Using Cython to Speed up Numerical Python Programs, Proceedings of MekIT'09, 2009.*

[Ramach] P. Ramachandran et al., *Performance of NumPy*, http://www.scipy.org/PerformancePython.

[Tutorial] S. Behnel, R. W. Bradshaw, D. S. Seljebotn, *Cython Tutorial*, Proceedings of the 8th Python in Science Conference, 2009.

[Cython] G. Ewing, R. W. Bradshaw, S. Behnel, D. S. Seljebotn, et al., *The Cython compiler*, http://cython.org.

[Pyrex] G. Ewing, *Pyrex: C-Extensions for Python*, http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/

[Python] G. van Rossum et al., *The Python language*, http://python.org.

[NumPy] T. Oliphant, http://numpy.scipy.org

[SciPy] E. Jones, T. Oliphant, P. Peterson, http://scipy.org

[f2py] P. Peterson, *f2py: Fortran to Python interface generator*, http://scipy.org/F2py.

[Weave] E. Jones, *Weave: Tools for inlining C/C++ in Python*, http://scipy.org/Weave.

[Instant] M. Westlie, K. A. Mardal, M. S. Alnæs, *Instant: Inlining of C/C++ in Python* http://fenics.org/instant.

[Psyco] A. Rigo, *Psyco: Python Specializing Compiler.* http://psyco.sourceforge.net.

[Sage] W. Stein et al., *Sage Mathematics Software*, http://sagemath.org/.

[BLAS] L. S. Blackford, J. Demmel, et al., *An Updated Set of Basic Linear Algebra Subprograms (BLAS)*, ACM Trans. Math. Soft., 28-2 (2002), pp. 135--151. http://www.netlib.org/blas/

[ATLAS] C. Whaley and A. Petitet, *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, in Software: Practice and Experience, 5, 2, 101-121, 2005, http://math-atlas.sourceforge.net/

[GCC] The GNU C Compiler, http://gcc.gnu.org/.

[ICC] The Intel C Compiler, http://software.intel.com/en-us/intel-compilers/.

[mpi4py] L. Dalcin et al., *mpi4py: MPI bindings for Python*, http://code.google.com/p/mpi4py/.

[OpenMPI] Open MPI, http://open-mpi.org

[Docs] http://docs.cython.org

[directives] http://wiki.cython.org/enhancements/compilerdirectives

[CPyAPI] Python/C API Reference Manual, http://docs.python.org/c-api/

[CPyPickle] The Python pickle module. http://docs.python.org/library/pickle.html

[PEP3118] T. Oliphant, C. Banks, *PEP 3118 - Revising the buffer protocol*, http://www.python.org/dev/peps/pep-3118.

[GSoC] The Google Summer of Code Program, http://code.google.com/soc.

High-Performance Code Generation Using CorePy

Andrew Friedley (afriedle@indiana.edu) – *Indiana University,* USA
Christopher Mueller (chemuell@indiana.edu) – *Indiana University,* USA
Andrew Lumsdaine (lums@indiana.edu) – *Indiana University,* USA

Although Python is well-known for its ease of use, it lacks the performance that is often necessary for numerical applications. As a result, libraries like NumPy and SciPy implement their core operations in C for better performance. CorePy represents an alternative approach in which an assembly language is embedded in Python, allowing direct access to the low-level processor architecture. We present the CoreFunc framework, which utilizes CorePy to provide an environment for applying element-wise arithmetic operations (such as addition) to arrays and achieving high performance while doing so. To evaluate the framework, we develop and experiment with several ufunc operations of varying complexity. Our results show that CoreFunc is an excellent tool for accelerating NumPy-based applications.

Python is well-known for its ease of use, and has an excellent collection of libraries for scientific computing (e.g., NumPy, SciPy). However, Python does not have the performance that is necessary for numerical applications. Many Python libraries work around this by implementing performance-critical functionality in another language. NumPy for example implements many of its operations in C for better performance. Furthermore, modern processor architectures are adding functionality (e.g. SIMD vector extensions, highly specialized instructions) that cannot be easily exploited in existing languages like C or Python.

CorePy is a tool (implemented as a Python library) aimed at providing direct access to the processor architecture so that developers can write high-performance code directly in Python.

To make CorePy more accessible to scientific applications, we have developed a framework for implementing NumPy ufunc operations (element-wise arithmetic operations extended to arrays) using CorePy. Although some performance gains can be had for the existing ufunc operations, we have found that our framework (referred to as CoreFunc) is most useful for developing custom ufuncs that combine multiple operations and take advantage of situation-specific optimizations to obtain better performance. To evaluate our framework, we use examples ranging from addition, to vector normalization, to a particle simulation kernel.

Accelerated NumPy Ufuncs

NumPy is a Python library built around an advanced multi-dimensional array object. Simple arithmetic such as addition and multiplication are supported as element-wise operations over the array class; these operations are referred to as ufuncs. Ufunc operations are implemented in C and invoked directly from Python. Although ufuncs are a powerful and convenient tool for working with data in arrays, they incur significant performance overhead: computational kernels are built using many ufunc calls, each of which makes a pass over its input arrays and sometimes allocates a new temporary array to pass on to later operations. When the input/output arrays do not fit entirely into cache, data must be transferred multiple times to and from memory, creating a performance bottlneck.

The Numexpr library [Numexpr], part of the SciPy project, addresses this problem by evaluating an arithmetic expression (e.g. $a + b * c$) expressed as a string. The arrays to be processed are broken down into smaller blocks which fit entirely into the processor's cache. An optimized loop is generated (in C) to perform the complete operation on one block before moving on to the next. A limitation of this approach however, is that Numexpr relies on a compiler to take advantage of advanced processor features (e.g., SIMD vector extections). Although modern compilers are always improving, this approach does not give the user the opportunity to introduce their own optimizations that a compiler may not be capable of implementing.

To address the shortcomings of these existing approaches, we developed the CoreFunc framework, in which CorePy is used as a backend for writing custom ufunc operations. Our primary goal in developing this framework was to create a system not just for accelerating the existing ufunc operations, but also to allow arbitrary custom operations to be defined to fully optimize performance. By using CorePy, the CoreFunc framework makes the entire processor architecture (e.g., vector SIMD instructions) available for use in a ufunc operation. Furthermore, we take advantage of multi-core functionality in recent processors by splitting work across multiple threads. Each thread runs in parallel, invoking the ufunc operation over a sub-range of the arrays.

A seconday goal of the framework is to reduce the effort needed to implement optimized ufunc operations. A CoreFunc user needs only to write code specific to their operation. This is a distinct advantage over using a C module (and perhaps inline assembly) to implement custom ufunc operations, in which case a user must also implement and debug a significant amount of auxiliary code, distracting from the task at hand.

The CoreFunc framework interface consists of only two functions. The first, `gen_ufunc`, generates the code to perform a ufunc operation on a specific datatype. We require that a separate code segment be generated for each data type the ufunc should support. The implementation of the operation can vary greatly from type

to type due to varying processor capabilities and available instructions. Three code segments are needed for a ufunc operation. The first is a vector/unrolled loop body, which operates on multiple elements per loop iteration, and performs the majority of the work. When fewer elements need to be processed than are handled by a single iteration of the vector loop body, a second scalar loop body is used to process one element per ieration. Lastly, a reduction operation is needed, and is usually just a single instruction. `gen_ufunc` generates the surrounding loop initialization, flow control code, ands atomic reduction operations to support multicore parallelism. A synthetic program is returned that can be invoked directly by NumPy to perform an operation on a specific array and data type.

Once the code for a ufunc operation has been generated, the `create_ufunc` is used to create a complete ufunc object. Synthetic programs for each data type to be supported by the ufunc are combined into a single, invocable ufunc object. Rather than calling synthetic programs directly, a C-based wrapper function is introduced to split work among multiple threads to run on multiple cores.

Ufuncs created using the CoreFunc framework behave very similarly to NumPy's built-in ufuncs. The following example shows the use of both the NumPy and CoreFunc addition implementations:

```
>>> import numpy
>>> import corefunc

>>> a = numpy.arange(5, dtype=numpy.int32)
>>> b = numpy.arange(5, dtype=numpy.int32)

# NumPy ufunc
>>> numpy.add(a, b)
array([ 0,  2,  4,  6,  8], dtype=int32)

# CorePy ufunc
>>> corefunc.add(a, b)
array([ 0,  2,  4,  6,  8], dtype=int32)
```

Reduction works in the same way, too:

```
>>> corefunc.add.reduce(a)
10
```

CorePy

Before evaluating applications of the CoreFunc framework, an a brief introduction to CorePy is necessary. The foundation of CorePy is effectively an object-oriented assembler embedded in Python; more advanced and higher-level abstractions are built on top of this to assist with software development. Assembly-level elements such as Instructions, registers, and other processor resources are represented as first-class objects. These objects are combined and manipulated by a developer to generate and transform programs on the fly at run-time. Machine-level code is synthesized directly in Python; no external compilers or assemblers are required.

The following is a simple example that defines a *synthetic program* to compute the sum $31+11$ and returns the correct result:

```
# Create a simple synthetic program
>>> prgm = x86_env.Program()
>>> code = prgm.get_stream()

# Synthesize assembly code
>>> code += x86.mov(prgm.gp_return, 31)
>>> code += x86.add(prgm.gp_return, 11)
>>> prgm += code

# Execute the synthetic program
>>> proc = Processor()
>>> result = proc.execute(prgm)
>>> print result
42
```

The first line of the example creates an empty `Program` object. `Program` objects manage resources such as register pools, label names, and the code itself. The code in a synthetic program consists of a sequence of one or more `InstructionStream` objects, created using the `Program`'s `get_stream` factory method. `InstructionStream` objects (effectively code segments) are containers for instructions and labels. The x86 `mov` instruction is used to load the value `31` into the special `gp_return` register. CorePy returns the value stored in this register after the generated program is executed. Next, and `add` instruction is used to add the value `11` to the return register. With code generation completed, the instruction stream is added into the program. Finally, a `Processor` object is created and used to execute the generated program. The result, `42`, is stored and printed.

CorePy is more than just an embedded assembler; a number of abstractions have been developed to make programming easier and faster. *Synthetic Expressions* [CEM06] overload basic arithmetic operators such that instead of executing the actual arithmetic operation, assembly code representing the operation is generated and added to an instruction stream. Similarly, *synthetic iterators* [CEM06] can be used to generate assembly-code loops using Python iterators and for-loop syntax. Specialized synthetic iterators can automatically unroll, vectorize, or parallelize loop bodies for better performance.

General code optimization is possible using an instruction scheduler transformation [AWF10] (currently only available for Cell SPU architecture). In addition to optimizing individual instruction streams, the instruction scheduler can be used to interleave and optimize multiple independent streams. For example, the sine and cosine functions are often called together (e.g., in a convolution algorithm). The code for each function can be generated separately, then combined and optimized to achieve far better performance. A similar approach can be used for optimizing and interleaving multiple iterations of an unrolled loop body.

Evaluation

To evaluate the effectiveness of the CoreFunc framework we consider two ufunc operations, addition and vector normalization. In addition, we implement the computational kernel portion of an existing particle simulation as a single ufunc operation.

Addition Ufunc

We implemented the addition ufunc to provide a direct comparison to an existing NumPy ufunc and demonstrate how the framework is used. Below is the CoreFunc implementation for 32-bit floating point addition:

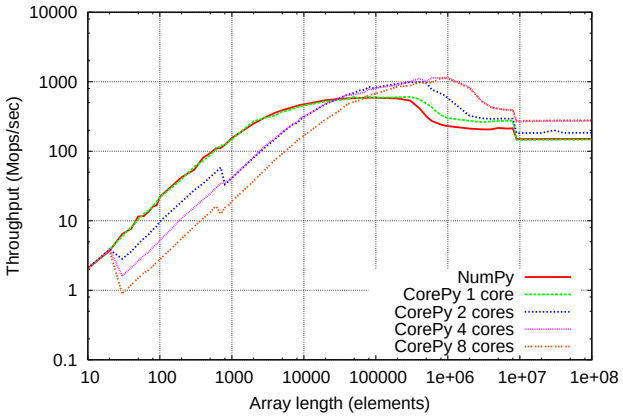
```
def gen_ufunc_add_float32():
  def vector_fn(prgm, r_args):
    code = prgm.get_stream()

    code += x86.movaps(xmm0, MemRef(r_args[0]))
    code += x86.addps(xmm0, MemRef(r_args[1]))
    code += x86.movntps(MemRef(r_args[2]), xmm0)
    return code

  def scalar_fn(prgm, r_args):
    code = prgm.get_stream()

    code += x86.movss(xmm0, MemRef(r_args[0]))
    code += x86.addss(xmm0, MemRef(r_args[1]))
    code += x86.movss(MemRef(r_args[2]), xmm0)
    return code

  return gen_ufunc(2, 1, vector_fn, scalar_fn, 4,
      x86.addss, ident_xmm_0, XMMRegister, 32)
```

First, note that very little code is needed in addition to the instructions that carry out the actual operation. The CoreFunc framework abstracts away unnecessary details, leaving the user free to focus on implementing their ufunc operations.

SIMD (Single-Instruction Multiple-Data) instructions simultaneously perform one operation on multiple values (four 32-bit floating point values in this case). x86 supports SIMD instructions using SSE (Streaming SIMD Extensions). In the above example, the `movaps` and `movntps` instructions load and store four contiguous floating point values to/from a single SIMD register. The `addps` instruction then performs four additions simultaneously. `scalar_fn` uses scalar forms of the same instructions to process one element at a time, and is used when there is not enough work remaining to use the vector loop. The `gen_ufunc` call generates and returns the CorePy synthetic program, which is later passed to `create_ufunc`.

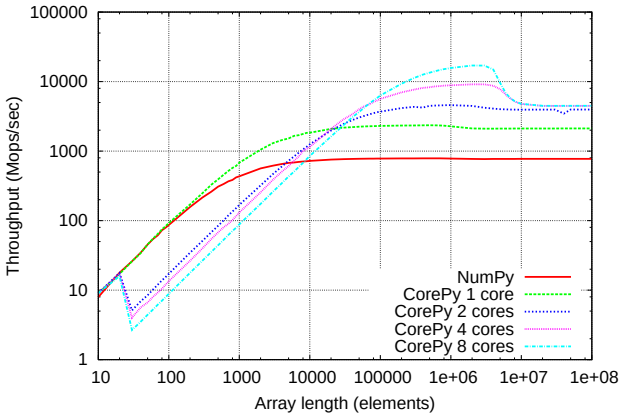
Timings were taken applying the ufuncs to varying array sizes. The average time to execute a varying number of iterations (80 to 10, scaling with the array length) at each array size was used to compute the number of ufunc operations completed per second. The system used contains two 1.86GHz Intel quad-core processors with 4mb cache and 16gb of ram, running Redhat Enterprise Linux 5.2. Python v2.6.2 and NumPy v1.3.0 were used.

Below, we compare the performance of NumPy's addition ufunc to the CoreFunc implementation using varying numbers of cores. Higher results are better.



Single-core performance using CoreFunc is highly similar to NumPy-a simple operation like addition is memory bound, so computational differences in implementation have little impact. Due to vector size and alignment requirements, multiple threads/cores are not used for array sizes less than 32 elements. We believe the performance gain realized using multiple cores is due to effectively having a larger processor cache for the array data and increased parallelism in memory requests, rather than additional compute cycles.

NumPy supports reduction as a special case of a normal ufunc operation in which one of the input arrays and the output array are the same, and are one element long. Thus the same loop is used for both normal operations and reductions. CoreFunc generates code to check for the reduction case and execute a separate loop in which the accumulated result is kept in a register during the operation. If multiple cores are used, each thread performs the reduction loop on a separate part of the input array, then atomically updates a shared accumulator with its part of the result. The performance comparison is below.



Vector Normalization Ufunc

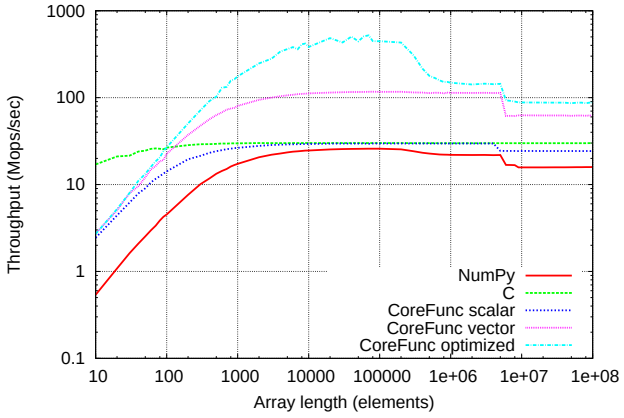
The second operation we examine is 2-D vector normalization, which was chosen as a more complex operation to experiment with how custom ufunc operations can achieve higher performance. The idea is that we

can combine operations to eliminate temporary arrays, and create opportunities for optimizations to better utilize the processor. Using NumPy, vector normalization is implemented in the following manner:

```
def vector_normalize(x, y):
  l = numpy.sqrt(x**2 + y**2)
  return (x / l, y / l)
```

Two arrays are taken as input, containing the respective x and y components for the vectors. Two arrays are output with the same data organization.

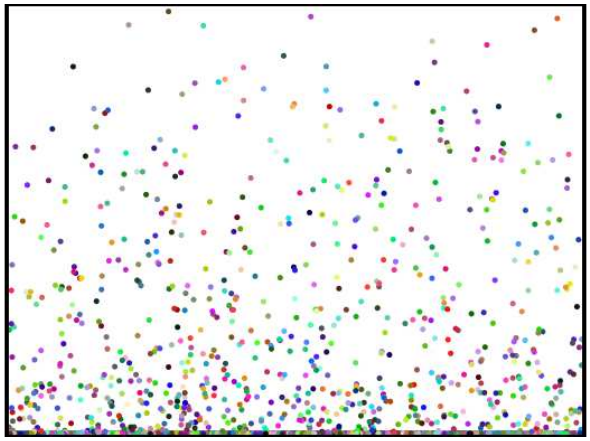
In the figure below, we compare the NumPy implementation, a C implementation (compiled using full optimizations with GCC 4.0.2), and the CoreFunc implementation with progressive stages of optimization. A single core is used for the CoreFunc results in this example. We believe the NumPy and CoreFunc implementations incur an overhead due to Python-level function calls to invoke the ufuncs, as well as type/size checking performed by NumPy. Otherwise, the C and CoreFunc scalar implementations are highly similar. NumPy is slightly slower due to additional overhead--each NumPy ufunc operation makes a pass over its entire input arrays and creates a new temporary array to pass to the next operation(s), rather than doing all the processing for one element (or a small set of elements) before moving on to the next.



We took advantage of CorePy's wide open processor access to progressively optimize the CoreFunc implementation. The scalar implementation shown in the figure is very similar to the assembly generated by the C code. SSE is used, but only scalar instructions--the code does not take advantage of SIMD vectorization. The CoreFunc vector implementation does however, resulting in significant performance gains. The same instruction sequence as the scalar implementation was used; we merely switched to the vector forms of the same instructions. Finally, we optimized the instruction sequence itself by taking advantage of the SSE reciprocal square-root function. This instruction is significantly less accurate than the square-root instruction, so we implemented a single iteration of the Newton-Raphson algorithm as suggested in [AMD09] section 9.11 to increase accuracy. [AMD09] suggests that this approach is not IEEE-754 compliant, but that the results are acceptable for most applications. Not only is this approach quicker than a single square-root instruction, we can also take advantage of the reciprocal square-root to convert the two division operations to multiplication for even more performance.

Particle Simulation

A particle simulation application has been used in previous work [CEM06b] to show how CorePy can be used to obtain speedups over a NumPy-based computational kernel. To experiment with using the CoreFunc framework to implement an entire kernel inside a single ufunc operation, we revisited the particle simulation application. Below is a screenshot of the running application:



The window forms a bounding box in which particles are subjected to gravity and air resistance. If a particle collides with any of the sides, its velocity in the appropriate axis is reversed, creating a bouncing effect. Particles are created interactively by moving the mouse pointer inside the window; their initial velocity is determined based on the velocity of the mouse movement.

A set of four arrays is used to represent the particles. Two arrays contain the x and y components of the particle positions, while the other two arrays contain the x and y components of the particle velocities. A fixed length for the arrays limits the number of particles on the screen; when the maximum number of particles is reached, the oldest particle is replaced.

The simulation is organized into a series of timesteps in which each particle's velocity and position is updated. A single timestep consists of one execution of the computational kernel at the core of the simulation. First, the particles' velocities are updated to account for forces imposed by gravity and air resistance. The updated velocity is then used to update each particle's position. Next, collision detection is performed. If any particle has moved past the boundaries of the window, its velocity component that is normal to the crossed boundary is negated. Bounces off the bottom edge, or floor, are dampened by scaling the value of the velocity y component. Implementation of the simulation in NumPy is straightforward; ufuncs and supporting

functions (e.g., `where` for conditionals) are used. Temporary arrays are avoided when possible by using output arguments to NumPy functions.

Our CoreFunc-based implementation moves all of the particle update calculations into a ufunc operation; we do this to evaluate whether this approach is a viable solution for developing high-performance computational kernels. The main loop uses SSE instructions to update four particles at a time in parallel. The four arrays are read and written only once per timestep; temporary values are stored in registers. Collision detection requires conditional updating of particle values. We achieved this without branches by using SSE comparison and bitwise arithmetic instructions. The following code performs collision detection for the left wall using NumPy:

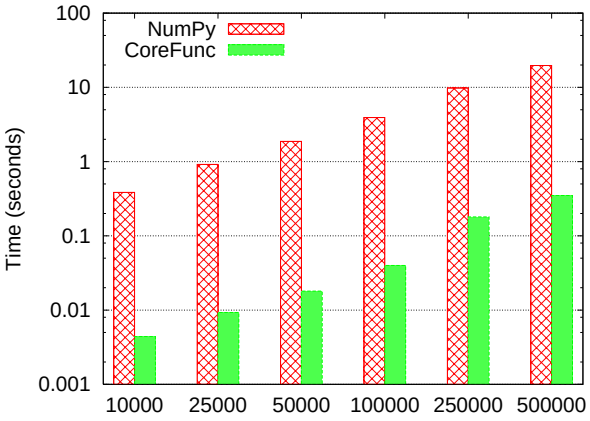
```
lt = numpy.where(numpy.less(pos_x, 0), -1, 1)
numpy.multiply(vel_x, lt, vel_x)
```

An optimized equivalent using SSE instructions looks like the following:

```
x86.xorps(r_cmp, r_cmp)
x86.cmpnltps(r_cmp, r_pos_x)
x86.andps(r_cmp, FP_SIGN_BIT)
x86.orps(r_vel_x, r_cmp)
```

The first instruction above clears the temporary comparison register by performing an exclusive-OR against itself (this is a common technique for initializing registers to zero on x86). The next instruction compares, in parallel, the x components of the positions of four particles against zero. If zero is *not* less than the x component, all 1 bits are written to the part of the comparison register corresponding to the x component. Otherwise, 0 bits are written. To make the particles bounce off the wall, the x velocity component needs to be forced to a positive value. However, only those particles that have crossed the wall should be modified. The 'truth mask' generated by the compare instruction is bitwise AND'd with a special constant with only the most significant bit (the floating point sign bit) of each value set. This way, only those particles whose direction needs to be changed are updated; particles that have not crossed the boundary will have a corresponding value of zero in the temporary comparison register. A bitwise OR operation then forces the sign bit of the velocity x component to positive for only those particles which have crossed the wall. Similar code sequences are used for the other three walls.

Implementing the simulation using CoreFunc proved to be straightforward, although getting the sequence of bit-wise operations right for collision detection took some creative thought and experimentation. The assembly code has surprisingly direct mapping from the NumPy code: this suggests that developing a set of abstractions for generating code analogous to common NumPy functions (i.e., `where`) would likely prove useful. We compare the performance of the two implementations in the following chart:



Timings were obtained by executing 100 timesteps in a loop. The display code was disabled so that only the simulation kernel was benchmarked. The system used was a 2.33GHz Intel quad-core processor with 4Mb cache. The operating system was Ubuntu 9.04; distribution-provided builds of Python 2.5.2 and NumPy 1.1.1 were used. Lower results indicate better performance.

Both implementations scale linearly with the number of particles, but the CoreFunc implementation is as much as two orders of magnitude (100x) faster. Even though the CoreFunc effort took more time to implement, this time pays off with a significant speedup. We conclude that our approach is suitable for development of simple to fairly complex computational kernels.

Conclusion

We have introduced the CoreFunc framework, which leverages CorePy to enable the development of highly optimized ufunc operations. Our experimentation with ufuncs of varying complexity shows that our framework is an effective tool for developing custom ufunc operations that implement significant portions of a computational kernel. Results show that the most effective optimizations are realized by combining simple operations together to make more effective use of low-level hardware capabilities.

Complete source code for CorePy, the CoreFunc framework, and a collection of sample ufunc implementations (including addition, vector normalization, and particle simulation) are available via anonymous Subversion at `www.corepy.org`. Code is distributed under BSD license.

Acknowledgements

Laura Hopkins and Benjamin Martin assisted in editing this paper. This work was funded by a grant from the Lilly Endowment, and by the Google Summer of Code program. Many thanks to Chris Mueller and Stefan van der Walt, who supported and mentored the CoreFunc project.

[AMD09] Advanced Micro Devices (AMD). Software Optimization Guide for AMD64 Processors, September 2005. http://developer.amd.com/documentation/guides/ (Accessed November 2009)

[Numexpr] D. Cooke, F. Alted, T. Hochberg, I. Valita, and G. Thalhammer. Numexpr: Fast numerical array expression evaluator for Python and NumPy. http://code.google.com/p/numexpr/ (Accessed October 2009)

[AWF10] A. Friedley, C. Mueller, and A. Lumsdaine. Enabling code transformation via synthetic composition. Submitted to CGO 2010.

[CEM06] C. Mueller and A. Lumsdaine. Expression and loop libraries for high-performance code synthesis. In LCPC, November 2006.

[CEM06b] C. Mueller and A. Lumsdaine. Runtime synthesis of high-performance code from scripting languages. In DLS, October 2006

Convert-XY: type-safe interchange of C++ and Python containers for NumPy extensions

Damian Eads (eads@soe.ucsc.edu) – *University of California, 1156 High Street, Santa Cruz* USA
Edward Rosten (er258@cam.ac.uk) – *University of Cambridge, Trumpington Street, Cambridge* UK

We present Convert-XY: a new, header-only template library for converting containers between C++ and Python with a simple, succinct syntax. At compile-time, template-based recursive pattern matching is performed on the static structure of the C++ type to build dynamic type checkers and conversion functions.

Note: proper C++ syntax encloses template parameters with angle brackets. We omit the extra space placed between two adjacent right brackets for space purposes, e.g. `vector<map<int, vector<int>>>` *instead of* `vector<map<int, vector<int> > >`.

Introduction

We present **Convert-XY**: a library for converting arbitrary containers between C++ and Python. Here is a simple "hello world" example of **Convert-XY** in action; it converts a Python dictionary `x` to a C++ object `y`:

```
void hello_world(PyObject *x) {
   map <string, vector<int>> y;
   convert(x, y);
}
```

This function can be called from Python with a dictionary (or any object obeying the mapping protocol) with key objects that can be converted to strings and sequence protocol objects containing objects that can be converted to numbers:

```
hello_world({"a": [1,2,3], "b": (4,5),
             "c": np.array([6,7]),
             "d": (x for x in (8,9))})
```

At compile time, pattern matching is applied to the structure of the C++ type of `y`. This yields two pieces of information: how to check for compatible structure in the Python type (e.g. a dictionary mapping strings to lists, tuples, arrays or sequences is compatible with `map<string, vector<int>>`) and which set of converters is applicable (e.g. sequences vs. dictionaries). Dynamic type checkers and converter functions can be built with this static type information at compile time. Further, the library statically determines when an array buffer pointer can be reused, preventing unnecessary copying which can be very useful for programs operating on large data sets. LIBCVD, a popular computer vision library used in our work, provides a `BasicImage` class, which can be used to wrap two-dimensional image addressing around a buffer pointer. **Convert-XY** avoids copying and allocation by reusing the NumPy buffer pointer whenever it can:

```
void example(PyObject *x) {
   map <string, BasicImage<float>> y;
   convert(x, y);
}
```

In this example, the type of the target object for conversion is known at compile time; it's an STL map. The compiler's template pattern matching algorithm matches on the structure of this type to build a dynamic type checker to ensure every part of the source Python object `x` follows compatible protocols, e.g. the outermost part of `x` follows the mapping protocol. In Python, we construct a dictionary of NumPy arrays containing integers.:

```
def example():
   hello = np.zeros((50,50), dtype='i')
   x = {'hello': hello}

   # Use your favorite wrapping tool to call C++
   # (e.g. Boost, Weave, Python C API)
   cpp.example(x);
```

Since the buffer is reused in this example and the NumPy array contains elements of a different type than the `BasicImage` objects in the STL map, a run time exception results, "`Expected a contiguous NumPy array of type code (f)`".
Unlike `BasicImage` objects, LIBCVD's `Image` objects allocate and manage their own memory. In this situation, the source `x` can be discontiguous and the array elements can be of a different type than in the source `x`.

Background

NumPy offers a succinct syntax for writing fast, vectorized numerical algorithms in pure Python. Low-level operations on large data structures happen in C, hidden from the user. The broader SciPy toolset (SciPy, NumPy, Mayavi, Chaco, Traits, Matplotlib, etc.) is a framework for developing scientific applications, simulations, GUIs, and visualizations [Jon01]. The simplicity and expressiveness of Python makes scientific computation more accessible to traditionally non-programmer scientists.
Writing extensions is tedious and inherently error-prone, but sometimes necessary when an algorithm can't be expressed in pure Python and give good performance. Additionally, a significant number of physics and computer vision codes are already written in C++ so Python/C++ integration is an unavoidable problem in need of good solutions.
Interfacing between C/C++ and Python can mean several different things; the figure illustrates three cases. First, wrapping C/C++ functions exposes their

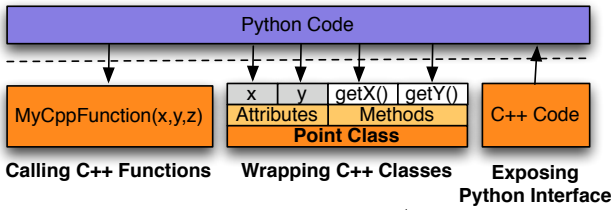


Figure 1 *Existing tools expose C/C++ functions, structures, and objects from Python and Python functions and data from C/C++.* ***Convert-XY*** *converts between arbitrary C++ and Python containers.*

interface so they can be called from Python. This is a well-explored problem with several mature tools in mainstream use including Boost Python [Abr03], SWIG [Bea95], Cython [Ewi08], Weave [Jon01], ctypes [Hel00], Py++ [Yak09], and the Python C API [GvR92] to name a few. Second, wrapping C++ structs and classes exposes user-defined data structures in Python. Boost, SWIG, and Py++ are very well-suited to interface problems of this kind. Third, exposing Python from C++ enables C++ code to manipulate Python objects. PyCXX serves this purpose. It wraps the Python Standard Library with C++ classes, performing reference counting and memory management automatically using C++ language features [Sco04].

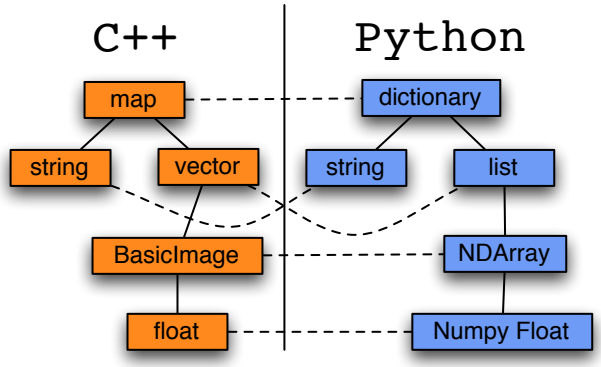


Figure 2 *Helper templates deduce sensible mappings and conversion behavior between Python and C++ types at compile time.*

Convert-XY is not meant to replace any of the other extension tools because it addresses a different problem altogether: how to convert objects between Python and C++ automatically.

SWIG supports typemaps which enable the developer to define how data is converted from Python to C++ and vice versa.:

```
namespace CVD {
  template<class T> class BasicImage;
  %typemap(in) BasicImage<float> {
     // Conversion code goes here.
  }
}
```

However the task of conversion must be specified manually. **Convert-XY** is a solution which can be used to automatically build converters for SWIG typemaps.

Differences in Type-checking

Python and C++ use fundamentally different type systems. In *dynamic typing* (Python), types are not prespecified and can change as the program runs. Thus, checks for type errors happen at *run time*. In *static typing* (C++), types never change throughout the execution of a program so type errors are found at compile time but type checks are omitted at run time because type correctness guarantees are made before the program starts. The template facility within C++ is a language in itself because templates are just compile time programs. Templates can be used to express sophisticated code generation and compile time logic. **Convert-XY** exploits the template facility to build conversion functions and dynamic type checkers for converting between C++ and Python objects at compile time.

Convert-XY

Convert-XY's names and functions are declared within the `ConvertXY` namespace. For the rest of this paper, we assume its symbols have been imported into the local namespace with `using namespace ConvertXY;`. We will be explicit when introducing a **Convert-XY** class or function for the first time.

The standard API for converting from a Python object to a C++ object is:

```
// From Python to C++
void ConvertXY::convert(PyObject *x, CPPType &y);
```

Reference counts are updated to ensure a consistent state in cases where the flow of execution returns to Python during conversion.

The `ConvertXY::convert` function is also used to convert from a C++ object back to a Python object; it returns an *unowned* reference to the Python object it creates:

```
// From C++ to Python
PyObject *ConvertXY::convert(CPPType &x);
```

In cases where an object of type `CPPType` does not support a default constructor, the `ConvertXY::Holder` class can be used:

```
void convert(PyObject *x, Holder<CPPType> &y);
```

The `Holder` class defers construction until more information is known about the source `x`, e.g. its dimensions, contiguity, or striding. A specialized `ConvertXY::ObjectFactory<CPPType>` class invokes a non-default constructor of `CPPType` with the information needed for construction. For example, an image size and buffer pointer are both needed to construct a `BasicImage` object. The `Holder` class conveniently enables us to declare a target without immediately constructing its innards. For example,

```
void example(PyObject *x) {
   Holder<BasicImage <float>> y;
   convert(x, y);
   BasicImage <float> &yr(y.getReference());
   display_image(yr);
}
```

The `BasicImage` object in the `Holder` object is not created until `convert` is called. The `getReference()` method returns a `BasicImage` reference created in `convert`. `Holder` 's destructor destroys the `BasicImage` object it contains. The `display_image` function opens up a viewer window to display the image. We assume that the control of execution never returns to Python while the viewer is still open. In a later section, we explain how to handle cases when this cannot be safely assumed.

`ToCPP` and `ToCPPBase` classes

`ConvertXY::ToCPPBase<CPPType, Action>` and `ConvertXY::ToCPP<CPPType, PyType, Action>` are the most fundamental classes in **Convert-XY**. Inheritence enables us to manage conversion between statically-typed C++ objects and dynamically-typed Python objects. The base class `ToCPPBase` assumes a C++ type but makes no assumptions about a Python type or protocol. Its derived class `ToCPP` assumes the Python object follows a protocol or has a specific type. There is a separate derived class for each possible (C++, Python) type pair. This effective design pattern mixes static polymorphism (templates) and dynamic polymorphism (inheritence) to properly handle two different typing paradigms. The `Action` template argument specifies how the conversion takes place; providing it is optional as an action specification is automatically generated for you by default.

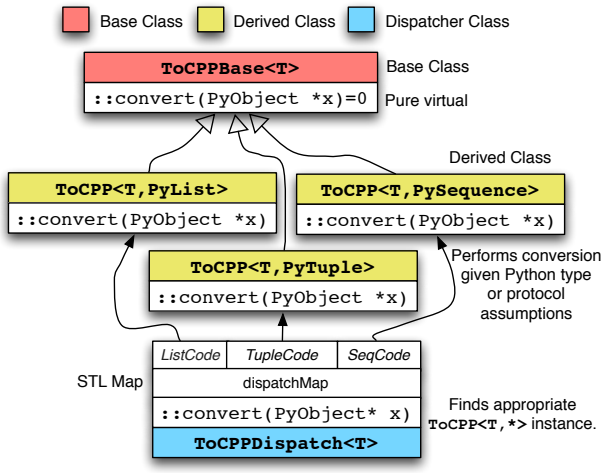


Figure 3 *The fundamental design pattern of* ***Convert-XY****: mixing dynamic polymorphism (inheritence) and static polymorphism (templates). The* `ToCPPDispatch` *class determines which types and protocols are applicable, and finds the appropriate* `ToCPP` *converter instance.*

`ConvertXY::ToCPPDispatch<CPPType, Action>` maintains an associative array between Python object types/ protocols and `ToCPP` derived classes. When the call `convert(x,y)` is compiled, pattern matching is applied on the type of `y` to recursively deduce which base converters could be needed at run-time. The `convert` function calls `ToCPPDispatch<CPPType,Action>::getConverter(x)`, which examines at run time the type and protocol information of the Python object `x` and returns a `ToCPP<CPPType,T,Action>` instance as a `ToCPPBase<CPPType>` reference, call it `z`. Then, the code `z.convert(x,y)` converts `x` to `y` with type assumptions of `x` (encoded with the type `T`) deduced at run-time.

At compile time, a Python object is converted to a C++ object of type `CPPType`, `ToCPP` classes are instantiated recursively for each subpart of `CPPType`, and this is repeated until no more containing types remain. The end result from this recursive instantiation is a compiled chain of converters for interchanging C++ objects of a specific type. Figure 4 illustrates how recursive instantiation works on a `map<string, vector<BasicImage<float>>>`.

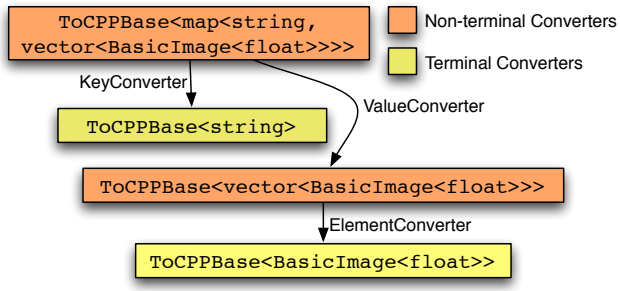


Figure 4 *Illustrates the process of instantiating* `Convert` *classes recursively. In orange are non-terminal converters which instantiate either non-terminal or terminal converters, and in yellow, terminal converters.*

In most cases, the basic user will never need to use to `ObjectFactory`, `ToCPP`, `ToCPPBase`, `ToCPPDispatch`, or `DefaultToCPPAction` classes directly.

Default Conversion Actions

Some C++ array classes, such as LIBCVD's `BasicImage`, do not manage their own memory and can reuse existing buffers such as ones coming from a different language layer (e.g. Python) or special memory from a device (e.g. a digital camera). These are called reference array classes. There are two different ways to convert a NumPy array to a reference array object: copy the buffer pointer or copy the data. `ConvertXY::DefaultToCPPAction` is a templated class for defining how each part of the conversion chain happens; it serves a purpose only at compile time. The `typedef`:

```
DefaultToCPPAction<map<string,
   vector<BasicImage<float>>>>::Action
```

is expanded recursively, mapping the STL `std::map` to the type:

```
Copy<Copy, Copy<Reuse>>
```

This compound type specifies that the string keys and vector elements be copied but the buffer pointer to the arrays should be reused. Figure 5 illustrates how this recursive instantiation works.

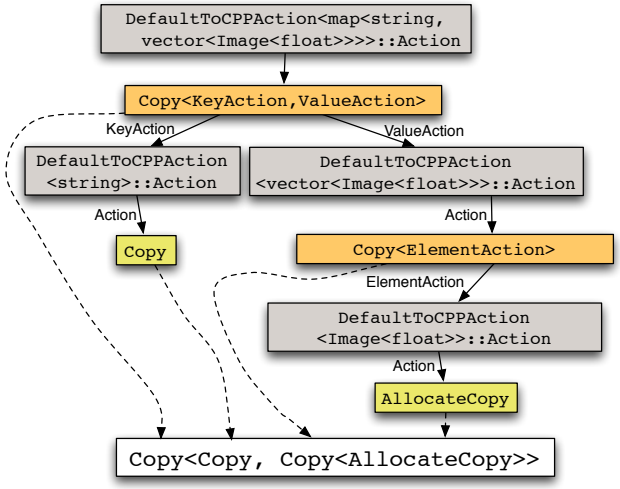


Figure 5 *The process of conversion is deduced at compile time via recursive instantiation of* `DefaultToCPPAction`*.*

The process of conversion can be overridden by passing as a template argument a different `Action` type. For example, suppose the target `BasicImage` was allocated from elsewhere (e.g. a special `malloc` function that properly aligns buffers), we can override the default conversion action (`Reuse`) so `Copy<>` is used as the `Action` instead. The `ConvertXY::convert_override` function can be used for this purpose:

```
void example(PyObject *x, BasicImage<float> &y) {
   convert_override<Copy<>>(x, y);
}
```

Suppose a C++ object `y` of a compound container type already exists, e.g. an object of type `map<string, BasicImage<float>>`. In this case, the keys should not be copied but we must ensure that the C++ map contains exactly the same set of keys as the Python dictionary. This is done by first checking that the size of the dictionary is the same as the `std::map` and then making sure each unique key in the dictionary is equivalent to some unique key in the map.

The contents of a dictionary of arrays can be copied into a `std::map` of existing `BasicImage` buffers as follows:

```
void example(PyObject *x,
             map<string, BasicImage<float>> &y) {
  convert_override
    <CheckSize<CheckExists, Copy<>>>(x, y);
}
```

The `CheckSize` simple checks that the size of the `std::map` and Python map are equivalent. `CheckExists` ensures each key in the Python map is also in the STL map.

Reference counting is automatically handled during conversion with PyCXX. In cases where the flow of execution may return to Python after conversion, PyCXX can be used to ensure objects that are in use by C++ routines don't get prematurely destroyed.:

```
void example(PyObject *x,
             BasicImage<float> &y) {

  // The line below is safe.
  convert(x,y);

  // However, if it can't be guaranteed that program
  // control flow will never return to Python
  // for the rest of this function, use PyCXX's
  // Py::Object to increment the reference count.
  Py::Object obj(x);


  // If the flow of execution returns to Python while
  // executing the function below, y will always point
  // to a valid buffer.
  compute(y);
}
```

The Other Way Around: from C++ to Python

When converting from C++ to Python, the type structure of the target Python object must be prespecified at compile time (e.g. the type of the Python object to be created is static). This is not straightforward since there is sometimes more than one compatible Python type for a C++ type. For example, an STL vector can be converted to a Python list, tuple, or NumPy object array. In **Convert-XY** this is deduced with a compile-time-only `Structure` metatype deduced by recursive instantiation of `DefaultToPythonStructure<CPPType>::Structure`. Figure 6illustrates how `DefaultToPythonStructure` is recursively instantiated to yield a default Python target type.

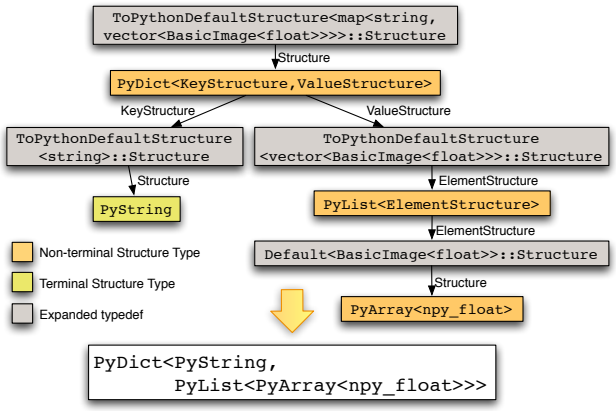


Figure 6 *The default Python type when converting from C++ to Python is deduced by recursive instantiation of* `ToPythonDefaultStructure<CPPType>`*.*

As the figure illustrates, the default type for a Python target given a source object that's an STL map mapping string to vectors of images is `PyDict<PyString, PyList<PyArray<npy_float>>>`. If a tuple is preferred over a list, the default structure can be overriden as follows:

```
PyObject *y
  = convert_override
      <PyDict<PyString,
              PyTuple<PyArray<npy_float>>>>(x)
```

If an attempt is made to convert a C++ object to a Python object that's incompatible, meaningful compiler messages can be generated via a trick involving incomplete types:

```
in not_found.hpp:88:
  'unsupport_type_or_structure'
  has incomplete type
  CannotConvertToPython<map<int,vector<int>>,PyInt>.
```

Allocating Result Arrays

One approach to allocating result arrays is to copy the result (e.g. image or matrix) into a new NumPy array. This overhead can be avoided using the `ConvertXY::allocate_numpy_array` function. This function is templated, and pattern matching is performed on the C++ type to generate an allocator at compile time. In this example, when the `allocate_numpy_array` function is called, NumPy array and `BasicImage` object are allocated at the same time, sharing the same buffer pointer. The image is smoothed using the LIBCVD function `convolveGaussian`, as illustrated on figure 7:

```
PyObject *smooth_image(PyObject *x, double radius) {

    // First convert the NumPy array input image
    // to a BasicImage<float>.
    Holder<BasicImage <float>> y;
    convert(x, y);

    // Now allocate enough memory to store the
    // result. Use the same buffer pointer for
    // both the NumPy array and BasicImage.
    Holder<BasicImage<float>> result;
    PyObject *pyResult =
      allocate_numpy_array(result,
       y.getReference().size());

    // Using LIBCVD, convolve a Gaussian on the
    // converted input, store the result in the
    // buffer pointed to by both pyResult and
    // result.
    CVD::convolveGaussian(y.getReference(),
                          result.getReference(),
                          radius);

    return pyResult;
}
```

Libraries Supported

Convert-XY supports conversion between Python and objects from several C++ libraries including TooN, LIBCVD, and STL. The library is split into three different headers, all of which are optional.

- `ConvertXY/TooN.hpp`: for converting between NumPy arrays and TooN matrices and vectors.
- `ConvertXY/STL.hpp`: for converting between STL structures and Python objects.

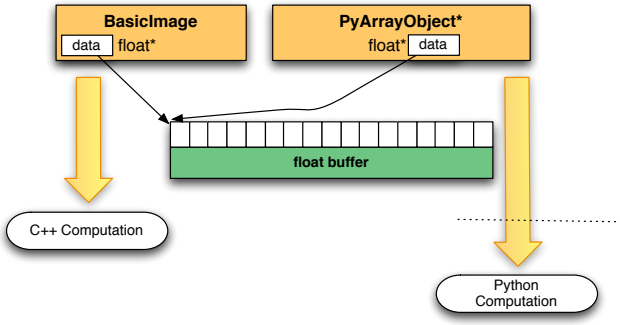


Figure 7 *`allocate_numpy_array` constructs a new Python and C++ array at the same time, to wit, a NumPy array and a `BasicImage` object. The NumPy array owns the buffer allocated so when its reference count goes to zero, its memory buffer will be freed.*

- `ConvertXY/CVD.hpp`: for converting between NumPy arrays and `CVD::Image` C++ objects.

The compiler can convert combinations of STL, CVD, and TooN structures (e.g. `vector<pair<BasicImage<float>,Matrix<>>>`) only when their respective headers are included together.

Tom's object-oriented Numerics Library (TooN)

TooN is a header-only linear algebra library that is very fast on small matrices making it particularly useful for real-time Computer Vision applications [Dru03]. TooN provides templated classes for static and dynamically-sized vectors and matrices, and uses templated-based pattern matching to catch for common linear algebra errors at compile time. For example, when converting the result of an a matrix multiplication involving two matrices `X` and `Y` of incompatible dimensions:

```
Matrix <3,5> X;
Matrix <6,3> Y;
PyObject *Z;
Z = convert(X * Y);
```

an easy-to-understand compiler error results, "`dimension_mismatch has incomplete type SizeMismatch<5,6>`".

Dimension checking is performed at compile time even when some dimensions are not known beforehand as the following example shows:

```
Matrix <Dynamic,5> X;
Matrix <6,Dynamic> Y;
PyObject *Z;
Z = convert(X * Y);
```

If a dimension check passes at compile time, it is omitted at run time. A buffer pointer of a NumPy array can be reused in a `TooN::Matrix` by specifying a layout type as the third template argument to `Matrix` (which is matched with `DefaultAction`):

```
void example(PyObject *x) {
  Holder<Matrix <Dynamic,
                 Dynamic,
                 Reference::RowMajor>> y;
  convert(x, y);
  cout << "The matrix is: "
       << y.getReference() << endl;
}
```

Calling the function above from Python with a very large NumPy array incurs practically no overhead because the NumPy array buffer pointer is copied.

Cambridge Video Dynamics Library (LIBCVD)

LIBCVD is a C++ library popular in the framerate, real-time computer vision community [Ros04]. The three main classes of the library are `BasicImage`, `Image`, and `ImageRef`. The base class `BasicImage` does not manage its own memory and can reuse buffer pointers from other objects such as NumPy arrays. The `Image` class inherits from the `BasicImage`, allocates its own memory, keeps a reference count of the number of `Image` objects referring to it, and deletes its buffer when the reference count drops to zero. `ImageRef` is simply a location in an image, i.e. an `(x,y)` pair. `Image` inherits from `BasicImage` so algorithms can be written to generically operate on both `Image` and `BasicImage` objects. Invoking `size()` on an image returns an `ImageRef`. The `ImageRef` objects are also used to index the image.

LIBCVD contains many of the most common image processing algorithms including convolution, morphology, camera calibration and connected components. It can load and process streaming video in many formats as well as PNG, JPEG, BMP, and PNM image formats.

The example below converts a NumPy array to a BasicImage, finds all local maxima within a 3 by 3 window above a threshold, converts the result to a Python object, and returns it:

```
template <class T>
PyObject* large_local_maxima(PyObject *pyImage,
                             PyObject *pyThresh) {
  Holder<BasicImage<T>> image;
  double thresh;

  convert(pyImage, image);
  convert(pyThresh, thresh);

  BasicImage<T> &I(image.getReference());

  vector<ImageRef> maxima;
  for(int y=1; y < I.size().y-1; y++)
    for(int x=1; x < I.size().x-1; x++) {
      T ctr = I[y][x];
      if(ctr > thresh      && ctr > I[y-1][x-1] &&
         ctr > I[y-1][x]   && ctr > I[y-1][x+1] &&
         ctr > I[y][x-1]   && ctr > I[y][x+1]   &&
         ctr > I[y+1][x-1] && ctr > I[y+1][x] &&
         ctr > I[y+1][x+1] ) {
        maxima.push_back(ImageRef(x, y));
      }
    }
  }
  return convert(maxima);
}
```

The example shows that with LIBCVD's lightweight, simple design, computer vision codes can be written in C++ succinct with good run-time performance. **Convert-XY** in combination with an existing wrapping tool greatly simplifies the task of interfacing Python with a C++ library.

The function is templated so it works on array elements of any numeric type. Templated functions are harder to wrap from Python. One simple, flexible approach involves writing a function that walks across a list of types specified as a template argument. In this case, the list of types is called `List`:

```
template <class List>
PyObject*
   large_local_maxima_walk(PyObject *pyImage,
                           PyObject *pyThresh) {
  typedef List::Head HeadType;
  typedef List::Tail TailList;
  const bool listEnd = List::listEnd;
  if (listEnd) {
     throw Py::Exception("type not supported!");
  }
  try {
     return large_local_maxima<HeadType>(pyImage,
                                         pyThresh);
  }
  catch (ConvertXY::TypeMismatch &e) {
     return large_local_maxima_walk<TailList>(pyImage,
                                              pyThresh);
  }
}
```

Then write a C wrapper function:

```
extern "C"
PyObject* large_local_maxima_c(PyObject *pyImage,
                               PyObject *pyThresh) {
  return large_local_maxima_walk
     <ConvertXY::NumberTypes>(pyImage, pyThresh);
}
```

that calls the walk function on a list of types. This C function can easily be called with SWIG, the Python C API, PyCXX, boost, or Cython. When debugging new templated functions, each error is usually reported by the compiler for each type in the type list. This can be avoided by first debugging with a list containing only a single type:

```
extern "C"
PyObject* large_local_maxima_c(PyObject *pyImage,
                               PyObject *pyThresh) {
  return large_local_maxima_walk
         <ConvertXY::SingletonList<float>>
            (pyImage, pyThresh);
}
```

Improvements and PyCXX Integration

Convert-XY has been refactored considerably since its first introduction at the SciPy conference in August 2009. The most significant improvements include customizable actions and structures, greatly simplified semantics, as well as being fully integrated with the mature C++ package PyCXX. It handles Python referencing and dereferencing in **Convert-XY** is handled to make conversion safe to exceptions and return of execution control to Python.

Packages Using Convert-XY

The authors have written several packages that already use **Convert-XY** .

- `GGFE`: grammar guided feature extraction is a technique for generating image features from generative grammars. GGFE is a tool for expressing generative grammars in pure Python. (see http://ggfe.googlecode.com). Some image features are implemented in C++ using LIBCVD with conversion handled by **Convert-XY**.
- `ACD`: a Python package for anomalous change detection in hyperspectral imagery. All algorithms are written in pure Python but optional, performance-enhanced C++ versions of functions make heavy use of **Convert-XY**. `ACD` is property of the Los Alamos National Laboratory and its release is still being considered by management.
- `ETSE`: a package for performing dynamic time warping and time series clustering in Python. (see http://www.soe.ucsc.edu/~eads/software.shtml)
- `Tesla`: a new object localisation system under development as part of our research. The software will be released upon publication of our algorithm.

Conclusion

Convert-XY is a powerful tool that facilitates automatic conversion of arbitrarily structured containers between C++ and Python with a succinct syntax, `convert(x,y)`. By exploiting template-based pattern matching in C++, dynamic type checkers and converters can be recursively built based on the static structure of a C++ object. At run-time, the dispatcher class decodes the type of the Python object and deduces the protocols it obeys. Additionally, conversion is customizable by specifying different action or structure meta-types. Large data sets can be converted between Python and C++ with minimal copying. When possible, erroneous conversions are caught at compile time but otherwise caught at run time. **Convert-XY** integrates with PyCXX to improve exception safety during conversion. It can also be used to automatically facilitate conversion for SWIG typemaps.

License

Convert-XY is offered under the terms of the General Public License version 2.0 with a special exception.

> As a special exception, you may use these files as part of a free software library without restriction. Specifically, if other files instantiate templates or use macros or inline functions from this library, or you compile this library and link it with other files to produce an executable, this library does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU General Public License.

Future Work

The primary focus of **Convert-XY**'s development until now has been on greatly improving the safety, simplicity, and flexibility of the interface. Moving forward, we plan to focus efforts on improving the documentation, finishing a regression test suite, and writing a tutorial on how to write custom converters for other libraries.

References

[Abr03] D. Abrahams. *Building Hybrid Systems with Boost Python.* PyCon 2003. 2003.

[Bea95] D. Beazley. *Simplified Wrapper and Interface Generator.* http://www.swig.org/. 1995--.

[Dru03] T. Drummond, E. Rosten, et al. *TooN: Tom's Object-oriented Numerics.* http://mi.eng.cam.ac.uk/~twd20/TooNhtml/. 2003.

[Ewi08] M. Ewing. *Cython.* 2008--.

[Hel00] T. Heller. *ctypes.* 2000--.

[Jon01] E. Jones, T. Oliphant, P. Peterson, et al. "SciPy: Open Source Scientific tools for Python". 2001--.

[GvR92] G. van Rossum. *Python.* 1991--.

[Ros04] E. Rosten, et al. *LIBCVD.* 2004--

[Sco04] B. Scott, P. Dubois. *Writing Python Extensions in C++ with PyCXX.* http://cxx.sf.net/. 2004--.

[Yak09] R. Yakovenko. Py++: Language Binding Project. 2009.

Parallel Kernels: An Architecture for Distributed Parallel Computing

P. A. Kienzle (pkienzle@nist.gov) – *NIST Center for Neutron Research, National Institute of Standards and Technology, Gaithersburg, Maryland 20899* USA
N. Patel (npatel17@umd.edu) – *Department of Materials Science and Engineering, University of Maryland, College Park, Maryland 20742* USA
M. McKerns (mmckerns@caltech.edu) – *Materials Science, California Institute of Technology, Pasadena, California 91125* USA

Global optimization problems can involve huge computational resources. The need to prepare, schedule and monitor hundreds of runs and interactively explore and analyze data is a challenging problem. Managing such a complex computational environment requires a sophisticated software framework which can distribute the computation on remote nodes hiding the complexity of the communication in such a way that scientist can concentrate on the details of computation. We present PARK, the computational job management framework being developed as a part of DANSE project, which will offer a simple, efficient and consistent user experience in a variety of heterogeneous environments from multicore workstations to global Grid systems. PARK will provide a single environment for developing and testing algorithms locally and executing them on remote clusters, while providing user full access to their job history including their configuration and input/output. This paper will introduce the PARK philosophy, the PARK architecture and current and future strategy in the context of global optimization algorithms.

Introduction

In this paper we present PARK, a flexible job management and parallel computation framework which is being developed as a part of DANSE project [Ful09]. PARK is a high level tool written in Python to provide the necessary tools for parallel and distributed computing [Bal89]. The heart of the system is yet another parallel mapping kernel, hiding the details of communication and freeing the end-user scientist to concentrate on the computational algorithm. The success of environments like Matlab and NumPy show that using an abstract interface using high level primitives such as vector mathematics, slow interpreted languages can achieve high performance on numerical codes. For parallel computations, Google introduced the Map-Reduce algorithm [Dea04], a robust implementation of master-slave parallelism where nodes could enter into and out of the computation. Individual map-reduce programs do not have to deal with the complexities of managing a reliable cluster environment, but can achieve fast robust performance on distributed database applications. The powerful map construct can be used equally well in many scientific computing problems.

A number of Python packages are addressing parts of this problem. PaPy[Cie09] is based on map constructs and directed acyclic graphs (DAGs) for scalable workflows to process data. Parallel Python[Von09] allows users to run functions remotely or locally. They provide code movement facilities to ease the installation and maintenance of parallel systems. IPython Parallel[Per09] gives users direct control of remote nodes through an interactive python console. They provide a load balancing map capability as well as supporting direct communication between nodes. Pyro [Jon09] is a python remote object implementation with a name service to find remote processes. It includes a publish-subscribe messaging system. PARK uses ideas from all of these systems.

Concept and architecture

PARK is a user-friendly job management tool that allows easy interaction with heterogeneous computational environments.

PARK uses a front end for submitting complex computational jobs to variety of computing environments. The backend computational service can use a dedicated cluster, or a shared cluster controlled by a batch queue. The front end allows the user to submit, kill, resubmit, copy and delete jobs. Like PaPy, jobs will be able to be composed as workflows. Unlike traditional batch processing systems, applications can maintain contact with the running service, receiving notification of progress and sending signals to change the flow of the computation. Communication from services to the client is handled through a publish-subscribe event queue, allowing multiple clients to monitor the same job. The client can signal changes to running jobs by adding messages to their own queues. Clients can choose to monitor the job continuously or to poll status periodically. The emphasis of the system is on early feedback and high throughput rather than maximum processing speed of any one job.

Remote jobs run as service components. A service component contains the full description of a computational task, including code to execute, input data for processing, environment set-up specification, post-processing tasks, output data produced by the application and meta-data for bookkeeping. The purpose of PARK can then be seen as making it easy for end-user scientist to create, submit and monitor the progress of services. PARK keeps a record of the services created

and submitted by the user in a persistent job repository.

The PARK API is designed for both the needs of interactive Graphical User Interfaces (GUI) and for scripted or Command Line Interfaces (CLI). A service in PARK is constructed from a set of components. All services are required to have an application component and a backend component, which define respectively the software to be run and the computational resources to be used. A client can connect to multiple backends at the same time, one of which will be the client machine. Services have input and output data components which are used during service execution. The overall PARK architecture is illustrated in figure 1. PARK monitors the evolution of submitted services and categorizes them into submitted, waiting, scheduled, running, completed, failed or killed states. All service objects are stored in a job repository database and the input and output files associated with the services are stored in file workspace. Both job repository and the file workspace may be in a local file system or on a remote server.

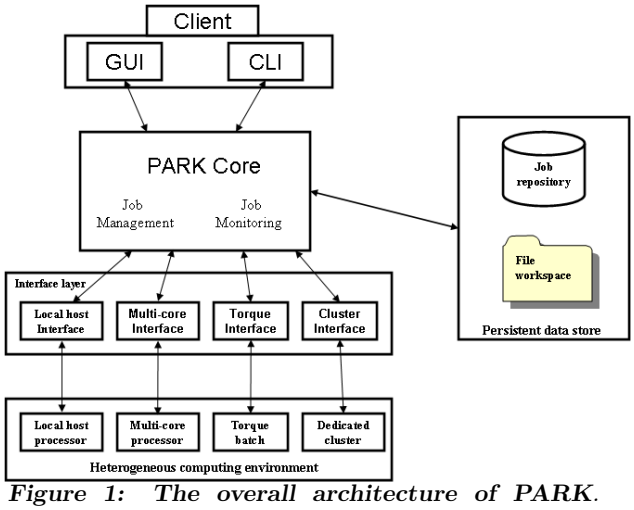


Figure 1: The overall architecture of PARK. *The client interacts with PARK via the Graphical User Interface (GUI) or via Command Line Interface (CLI). All jobs are stored in the persistent data store.*

PARK interface

The client initiates the connection to PARK through a connect object, with the URL of the resource as a parameter. Different resources may use different connection and authentication and transport protocols, but the returned connection object provides a unified interface. The connection may be to the local machine or a remote cluster. Client can submit their jobs through this connect object to respective computational resources and also retrieve the results of jobs. PARK provides functions for executing jobs, retrieving the results of finished jobs, checking the status of the jobs and controlling the jobs. The functions provided by PARK are independent of various computational resources so that a client may write a single program that will run on any platform with minor changes in interface.

The connect function creates a new connection object associated with specified computational resources. If no connection is established, then the job will be run on the client machine. The example below connects to the cluster at compufans.ncnr.nist.gov:

```
compufans = park.connect('compufans.ncnr.nist.gov')
```

The submit function creates a job and submits it to the compufans job queue. This function returns immediately. It returns job id back to the client with which client can control the job. The executable can be a script or a function. If executable is a function, the modules argument specify what modules are required to execute the function:

```
job_id = compufans.submit(executable,
                  input_data=None, modules=())
```

The get_result_id function returns the result of the submitted job after it is completed, or None if it is not completed:

```
result = compufans.get_result_id(job_id=None)
```

The get_status function returns a python dictionary with keys specifying the current status of all the three queues (waiting, running and finished) and values as number of jobs in each queues:

```
status_dict = compufans.get_status(tag)
```

Job management architecture

The job management components include job creator, job scheduler and job launcher. Its architecture is shown on figure 2.

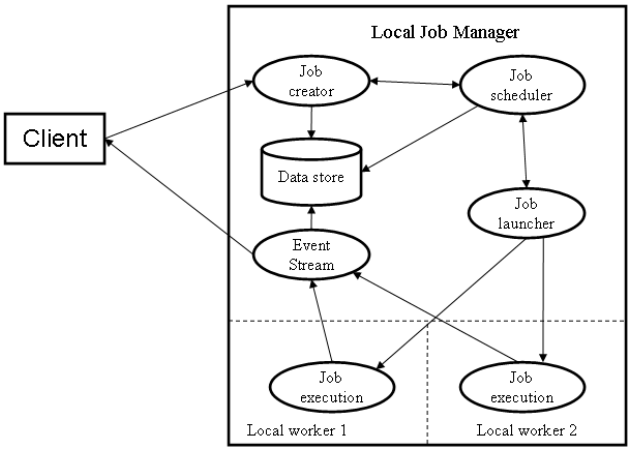


Figure 2: Job management architecture. *In this case client connects to a local system and PARK automatically identifies the number of cores present in the system and spawns that many workers to run job. The job communicates back to the client through a publish-subscribe event queue.*

The job creator receives information on the jobs from the client either through GUI or through CLI. The

job creator assigns an job identifier to every job and returns this to the client. The combination of job manager URL and job id is unique.

The job scheduler is the core scheduling component which decides which jobs to run when. The scheduler is responsible for maintaining enough information about the state of the jobs to make good decisions about job placement. All the resources are allocated to the jobs by job priority. This ensures that high-priority jobs are added at the front of the queue. If jobs have equal priority, resources are allocated to the job that was submitted first. The job scheduler also supports backfill. This ensures that a resource-intensive application will not delay other applications that are ready to run. The job scheduler will schedule a lower-priority job if a higher-priority job is waiting for resources to become available and the lower-priority job can be finished with the available resources without delaying the start time of the higher-priority job.

The job launcher's function is to accept the jobs and launch them. Job launcher contains the logic required to initiate execution on the selected platform for the selected network configuration. The job launcher receives messages sent by the job scheduler. When receiving an execution request, it will create a new thread in order to allow asynchronous job execution. After getting a job identifier (i.e. the handle to the job at the server side) as the response, the job launcher will send it to the corresponding thread of the job scheduler. The job handle will be used to monitor the job status and control the job.

The job monitor is used to monitor the submitted jobs. Information about the remote execution site, queue status and successful termination are collected. The job manager maintains a connection to the running job so that the client can steer the job if necessary. For example, a fitting service may allow the user to change the range of values for a fit parameter. This level of control will not be possible on some architectures, such as TeraGRID nodes that do not allow connections back to the event server.

Job execution workflow

In order to simplify the complexity of job management, various operations of job management are organized as a workflow. Within the workflow the job can be deleted or modified at any time. PARK manages three job queues namely the waiting, running and finished queues and jobs are stored in one of the three queues based on their status. The waiting queue keeps the un-scheduled jobs. The running queue keeps running jobs and the finished queue keeps finished jobs which are either completed successfully or failed. After the job is created, it is put on the waiting queue. When all dependencies are met the job description is submitted to the job launcher and the job is moved to the running queue. On job completion, a completion event is published on the job event queue. The job monitor subscribes to this queue, and moves the job to the finished queue when it receives the completion event. After the job reaches the finished queue, the results are ready to be retrieved by the client (see figure 3). All the logging and error messages are automatically saved. At the client side, multiple jobs can be defined at the same time, so there may exist more than one job workflow in the system. All jobs are independent of each other and identified by either job ID or client tags. Future versions will support workflow conditions, only launching a new job when the dependent jobs are complete.

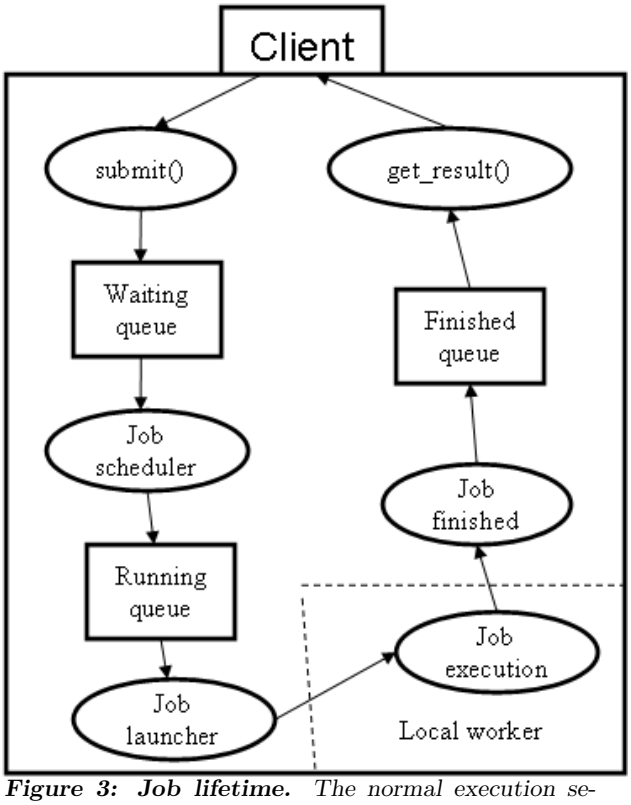


Figure 3: Job lifetime. *The normal execution sequence complete separates the client from the computational resources, allowing users to disconnect when the job is started and reconnect to get the results.*

Parallel mapper

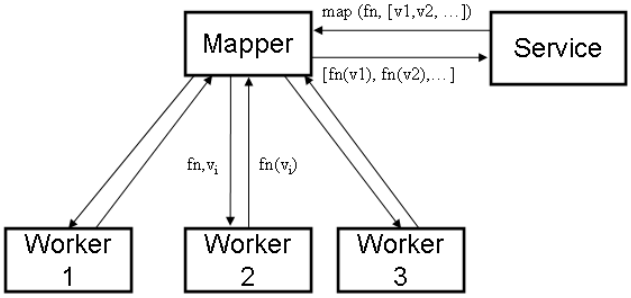


Figure 4: Mapper workflow architecture. *fn is the function to be mapped and v_i are the inputs. The function should be stateless, with output depending only on the current input v_i, not on the previous inputs.*

A running service has many facilities available. It can use a data store or file system for data retrieval. A service can use Mapper which applies a function to a set of inputs. Mapper acts like the built-in 'map' function of Python, but the work is completed in parallel across the available computing nodes. Mapper keeps tracks of all map requests from various running service and handles any issues like node failure or nested maps and infinite loops in a functions. Mapper can be excellent resource for data parallelism where same code runs concurrently on different data elements. Mapper workflow is described in figure 4, in which running service sends map request to Mapper via service handler. Mapper queues individual function-value pairs. Workers pull from this queue, returning results to Mapper. After all results are complete, Mapper orders them and sends them back to the service.

Global optimization

We are using PARK to construct a global optimization framework. The basic algorithm for optimization is as follows:

```
optimizer.configure()
population = optimizer.start(function.parameters)
while True:
    cost = map(function, population)
    optimizer.update(population)
    if converged():
        break
    post_progress_event()
    population = optimizer.next()
```

This algorithm applies equally to pure stochastic optimizers like differential evolution and genetic algorithms as well as deterministic methods such as branch and bound, and even quasi-Newton methods where multiple evaluations are required to estimate the descent direction.

In order to implement this algorithm in PARK, we need to define a Fit service which accepts an optimizer, some convergence conditions and the function to be optimized. The Fit service first registers the function with Mapper. Workers who participate in the map evaluation are first configured with the function, and then called repeatedly with members of the population. Since the configuration cost is incurred once per worker, we can handle problems with a significant configuration cost. For example a fit to a 100Mb dataset requires the file to be transferred to each worker node once at the beginning rather than each time the fitness function is evaluated. This mechanism also supports a dynamic worker pool, since new workers can ask Mapper for the function when they join the pool.

We have generalized convergence conditions. Rather than building convergence into each optimizer, the Fit service keeps track of the values of the population, the best fit, the number of iterations, the number of function calls and similar parameters that are used to control the optimizer. To write a new optimizer for PARK, users will need to subclass from Optimizer as follows:

```
class Optimizer:
    def configure(self):
        "prepare the optimizer"
    def start(self, parameters):
        "generate the initial population"
    def update(self, population):
        "record results of last population"
    def next(self):
        "generate the next population"
```

Each cycle the optimizer has the option of informing the client of the progress of the fit. If the fit value has improved or if a given percentage of the maximum number of iterations is reached then a message will be posted to the event stream associated with the fit job. If the cluster cannot connect to the event queue (which will be the case when running on TeraGRID machines), or if the client is not connected then this event will be ignored.

The assumption we are making is that the cost of function evaluations is large compared to the cost of generating the next population and sending it to the workers. These assumptions are not implicit to the global optimization problem; in some cases the cost of evaluating the population members is cheap compared to processing the population. In a dedicated cluster we can hide this problem by allowing several optimization problems to be run at the same time on the same workers, thus when one optimizer is busy generating the next population, the workers can be evaluating the populations from the remaining optimizers. A solution which can exploit all available parallelism will require dedicated code for each optimizer and more communication mechanisms than PARK currently provides.

Conclusions and future work

A robust job management and optimization system is essential for harnessing computational potential of various heterogeneous computing resources. In this paper we presented an overview of the architecture of PARK distributed parallel framework and its various features which makes running single job across various heterogeneous platform almost painless.

The PARK framework and the global optimizer are still under active development and not yet ready for general use. We are preparing for an initial public release in the next year.

In future we would like to implement new scheduling scheme for optimizing resource sharing. We will add tools to ease the monitoring of jobs and administering a cluster. We are planning various forms of fault tolerance features to PARK, making it robust against failure of any nodes, including the master node via checkpoint and restore. Security is also an important issue, further work being needed to improve the existing security features. Using the high level map primitive, ambitious users can implement PARK applications on new architectures such as cloud computing, BOINC[And04], and TeraGrid by specializing the map primitive for their system. The application code will remain the same.

Acknowledgments

This work is supported by National Science Foundation as a part of the DANSE project, under grant DMR-0520547.

References

[And04] D. P. Anderson. *BOINC: A system for public-resource computing and storage*, in R. Buyya, editor, 5th International Workshop on Grid Computing (GRID 2004), 8 November 2004, Pittsburgh, PA, USA, Proceedings, pages 4-10. IEEE Computer Society, 2004.

[Bal89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. *Programming languages for distributed computing systems* ACM computing Surveys, 21(3):261-322, September 1989.

[Cie09] M. Cieslik, *PaPy: Parallel and distributed data-processing pipelines in Python*, in Proceedings of the 8th Python in Science Conference (SciPy 2009)

[Dea04] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[Ful09] B. Fultz, et al.: *DANSE: Distributed data analysis for neutron scattering experiments*, http://danse.us/ (Accessed Aug. 2009)

[Jon09] I. de Jong: *Pyro - Python Remote Objects*, http://pyro.sourceforge.net/ (Accessed Aug. 2009)

[Per09] F. Perez and B. Granger: *IPython: a system for interactive scientific computing*, Computing in Science & Engineering 9(3) 21-29, 2007

[Van09] V. Vanovschi: *Parallel Python*, http://www.parallelpython.com/ (Accessed Aug. 2009)

PaPy: Parallel and distributed data-processing pipelines in Python

Marcin Cieślik (mpc4p@virginia.edu) – *University of Virginia,* U.S.
Cameron Mura (cmura@virginia.edu) – *University of Virginia,* U.S.

PaPy, which stands for parallel pipelines in Python, is a highly flexible framework that enables the construction of robust, scalable workflows for either generating or processing voluminous datasets. A workflow is created from user-written Python functions (nodes) connected by 'pipes' (edges) into a directed acyclic graph. These functions are arbitrarily definable, and can make use of any Python modules or external binaries. Given a user-defined topology and collection of input data, functions are composed into nested higher-order maps, which are transparently and robustly evaluated in parallel on a single computer or on remote hosts. Local and remote computational resources can be flexibly pooled and assigned to functional nodes, thereby allowing facile load-balancing and pipeline optimization to maximimize computational throughput. Input items are processed by nodes in parallel, and traverse the graph in batches of adjustable size - a trade-off between lazy-evaluation, parallelism, and memory consumption. The processing of a single item can be parallelized in a scatter/gather scheme. The simplicity and flexibility of distributed workflows using PaPy bridges the gap between desktop -> grid, enabling this new computing paradigm to be leveraged in the processing of large scientific datasets.

Introduction

Computationally-intense fields ranging from astronomy to chemoinformatics to computational biology typically involve complex workflows of data production or aggregation, processing, and analysis. Several fundamentally different *forms* of data - sequence strings (text files), coordinates (and coordinate trajectories), images, interaction maps, microarray data, videos, arrays - may exist in distinct file formats, and are typically processed using available tools. Inputs/outputs are generally linked (if at all) *via* intermediary files in the context of some automated build software or scripts. The recently exponential growth of datasets generated by high-throughput scientific approaches (*e.g.* structural genomics [TeStYo09]) or high-performance parallel computing methods (*e.g.* molecular dynamics [KlLiDr09]) necessitates more flexible and scalable tools at the consumer end, enabling, for instance, the leveraging of multiple CPU cores and computational grids. However, using files to communicate and synchronize processes is generally inconvenient and inefficient, particularly if specialized scientific Python modules (*e.g.*, BioPython [CoAnCh09], PyCogent [Knight07], Cinfony [OBHu08], MMTK [Hinsen00], Biskit [GrNiLe07]) are to be used.

Many computational tasks fundamentally consist of chained transformations of collections of data that are independent, and likely of variable type (strings, images, *etc.*). The scientific programmer is required to write transformation steps, connect them and - for large datasets to be feasible - parallelize the processing. Solutions to this problem generally can be divided into: *(i) Make*-like software build tools, *(ii)* workflow management systems (WMS), or *(iii)* grid engines and frontends. PaPy, which stands for parallel pipelines in Python, is a module for processing arbitrary streams of data (files, records, simulation frames, images, videos, *etc.*) *via* functions connected into directed graphs (flowcharts) like a WMS. It is not a parallel computing paradigm like *MapReduce* [DeGh08] or BSP [SkHiMc96], nor is it a dependency-handling build tool like Scons [Knight05]. Neither does it support declarative programming [Lloyd94]. In a nutshell, PaPy is a tool that makes it easy to structure procedural workflows into Python scripts. The tasks and data are composed into nested higher-order map functions, which are transparently and robustly evaluated in parallel on a single computer or remote hosts.

Workflow management solutions typically provide a means to connect standardized tasks *via* a structured, well-defined data format to construct a workflow. For transformations outside the default repertoire of the program, the user must program a custom task with inputs and outputs in some particular (WMS-specific) format. This, then, limits the general capability of a WMS in utilizing available codes to perform non-standard or computationally-demanding analyses. Examples of existing frameworks for constructing data-processing pipelines include Taverna (focused on web-services; run locally [OiAdFe04]), DAGMan (general; part of the Condor workload management system [ThTaLi05]) and Cyrille2 (focused on genomics; run on SGE clusters [Ham08]). A typical drawback of integrated WMS solutions such as the above is that, for tasks which are not in the standard repertoire of the program, the user has to either develop a custom task or revert to traditional scripting for parts of the pipeline; while such an approach offers an immediate solution, it is not easily sustainable, scalable, or adaptable, insofar as the *processing logic* becomes hardwired into these script-based workflows.

In PaPy, pipelines are constructed from Python functions with strict call semantics. Most general-purpose functions to support input/output, databases, inter-process communication (IPC), serialization, topology, and mathematics are already a part of PaPy. Domain-specific functions (*e.g.* parsing a specific file-format) must be user-provided, but have no limitations as to

functional complexity, used libraries, called binaries or web-services, *etc.* Therefore, as a general pipeline construction tool, PaPy is intentionally lightweight, and is entirely agnostic of specific application domains.

Our approach with PaPy is a highly modular workflow-engine, which neither enforces a particular data-exchange or restricted programming model, nor is tied to a single, specific application domain. This level of abstraction enables existing code-bases to be easily wrapped into a PaPy pipeline and benefit from its robustness to exceptions, logging, and parallelism.

Architecture and design

PaPy is a Python module "`papy`" written to enable the logical design and deployment of efficient data-processing pipelines. Central design goals were to make the framework *(i)* natively parallel, *(ii)* flexible, *(iii)* robust, *(iv)* free of idiosyncrasies and dependencies, and *(v)* easily usable. Therefore, PaPy's modular, object-oriented architecture utilizes familiar concepts such as *map* constructs from functional programming, and directed acyclic graphs. Parallelism is achieved through the shared worker-pool model [Sunderam90].

The architecture of PaPy is remarkably simple, yet flexible. It consists of only four core component *classes* to enable construction of a data-processing pipeline. Each class provides an isolated subset of the functionality [Table1], which together includes facilities for arbitrary flow-chart topology, execution (serial, parallel, distributed), user function wrapping, and runtime interactions (*e.g.* logging). The pipeline is a way of expressing **what** (*functions*), **where** (*toplogy*) and **how** (*parallelism*) a collection of (potentially interdependent) calculations should be performed.

Table 1: Components (*classes*) and their roles.

Component	Description and function
IMap[1]	Implements a process/thread pool. Evaluates multiple, nested map functions in parallel, using a mixture of threads or processes (locally) and, optionally, remote RPyC servers.
Piper Worker	Processing nodes of the pipeline created by wrapping user-defined functions; also, exception handling, logging, and scatter-gather functionality.
Dagger	Defines the data-flow and the pipeline in the form of a directed acyclic graph (DAG); allows one to add, remove, connect pipers, and validate topology. Coordinates the starting/stopping of IMaps.
Plumber	Interface to monitor and run a pipeline; provides methods to save/load pipelines, monitor state, save results.

Pipelines (see Figure 1) are constructed by connecting functional units (`Piper` instances) by directed pipes, and are represented as a directed acyclic graph data structure (`Dagger` instance). The pipers correspond to nodes and the pipes to edges in a graph. The topological sort of this graph reflects the input/output dependencies of the pipers, and it is worth noting that any valid DAG is a valid PaPy pipeline topology (*e.g.*, pipers can have multiple incoming and outgoing pipes, and the pipeline can have multiple inputs and outputs). A pipeline input consists of an iterable collection of data items, *e.g.* a list. PaPy does not utilize a custom file format to store a pipeline; instead, pipelines are constructed and saved as executable Python code. The PaPy module can be arbitrarily used within a Python script, although some helpful and relevant conventions to construct a workflow script are described in the online documentation.

The functionality of a piper is defined by user-written functions, which are Python functions with strict call semantics. There are no limits as to what a function does, apart from the requirement that any modules it utilizes must be available on the remote execution hosts (if utilizing RPyC). A function can be used by multiple pipers, and multiple functions can be composed within a single piper. CPU-intensive tasks with little input data (*e.g.*, MD simulations, collision detection, graph matching) are preferred because of the high speed-up through parallel execution.

Within a PaPy pipeline, data are shared as Python objects; this is in contrast to workflow management solutions (*e.g.*, Taverna) that typically enforce a specific data exchange scheme. The user has the choice to use any or none of the structured data-exchange formats, provided the tools for using them are available for Python. Communicated Python objects need to be serializable, by default using the standard Pickle protocol.

Synchronization and data communication between pipers within a pipeline is achieved by virtue of queues and locked pipes. No outputs or intermediate results are implicitly stored, in contrast to usage of temporary files by *Make*-like software. Data can be saved anywhere within the pipeline by using pipers for data serialization (*e.g.* JSON) and archiving (*e.g.* file-based). PaPy maintains data integrity in the sense that an executing pipeline stopped by the user will have no pending (lost) results.

Parallelism

Parallel execution is a major issue for workflows, particularly *(i)* those involving highly CPU-intensive methods like MD simulations or Monte Carlo sampling, or *(ii)* those dealing with large datasets (such as arise in astrophysics, genomics, *etc.*). PaPy provides

[1] Note that the IMap class is available as a separate Python module.

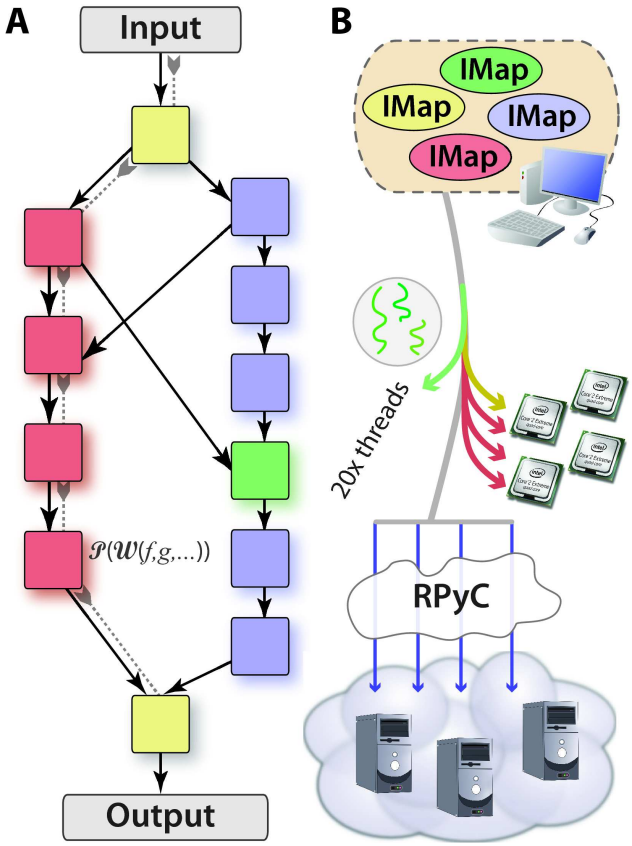


Figure 1. *(**A**) PaPy pipeline and its (**B**) computational resources. The directed graph illustrates the Dagger object as a container of Piper objects (nodes), connected by pipes (black arrows; in the upstream / downstream sense) or, equivalently, dependency edges (gray arrows). Pipers are assigned to various compute resources as indicated by different colors. The sets of pipes connecting the two processing streams illustrate the flexible construction of workflows. Encapsulation and composition of user-written functions e.g., f, g into a Worker and Piper object is represented as P(W(f,g,...)). Resources used by the sample pipeline are shown in B. Each virtual resource is an IMap object, which utilizes a worker pool to evaluate the Worker on a data item. IMaps are shared by pipers and might share resources. The resources are: a local pool of 20 threads used by a single piper in the pipeline (green); four CPU-cores, of which at most three are used concurrently (red) and one dedicated to handle the input/output functions (yellow); and a pool of Python processes utilizing remote resources exposedby RPyC servers (blue cloud). Parallelism is achieved by pulling data through the pipeline in adjustable batches.*

support for two levels of parallelism, which adress both of these scenarios: (1) parallel processing of independent input data items, (2) multiple parallel jobs for a single input item. The first type of parallelism is achieved by creating parallel pipers - *i.e.* providing an IMap instance to the constructor. Pipers within a pipeline can share an IMap instance or have dedicated computational resources (Fig. 1). The mixing of serial and parallel pipers is supported; this flexibility permits intelligent pipeline load-balancing and optimization. Per-item parallel jobs are made possible by the *produce / spawn / consume* (Fig. 2) idiom within a workflow. This idiom consists of at least three pipers. The role of the first piper is to produce a list of N subitems for each input item. Each of those subitems is processed by the next piper, which needs to be spawned N times; finally, the N results are consumed by the last piper, which returns a single result. Multiple spawning pipers are supported. The subitems are typically independent fragments of the input item or parameter sets. Per-item parallelism is similar to the *MapReduce* model of distributed computing, but is not restricted to handling only data structured as (*key*, *value*) pairs.

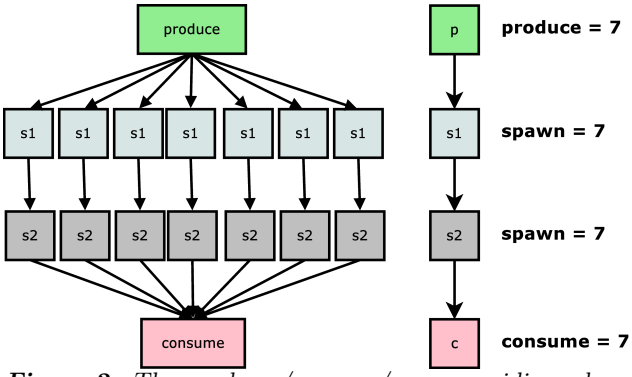


Figure 2. *The produce / spawn / consume idiom allows for parallel processing of a single input item in addition to parallel processing of items (explanation in text).*

The parallelism of an IMap instance is defined by the number of local and remote worker processes or threads, and the "stride" argument (Fig. 3), if it processes multiple tasks. The "stride" is the number of input items of task N processed before task $N+1$ commences. Tasks are cycled until all input items have been processed. In a PaPy pipeline pipers can share a computational resource; they are different tasks of a single IMap instance. The "stride" can also be considered as the number of input items processed by pipers in consecutive rounds, with the order defined by a topological sort of the graph. Therefore, the data traverses the pipeline in batches of "stride" size. A larger "stride" means that potentially more temporary results will have to be held in memory, while a smaller value may result in idle CPUs, as a new task cannot start until the previous one finishes its "stride". This adjustable memory/parallelism trade-off allows PaPy pipelines to process data sets with temporary results

too large to fit into memory (or to be stored as files), and to cope with highly variable execution times for input items (a common scenario on a heterogenous grid, and which would arise for certain types of tasks, such as replica-exchange MD simulations).

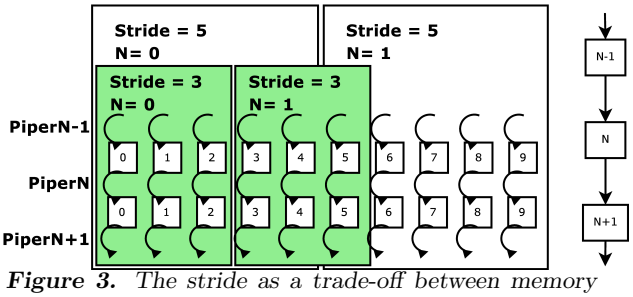


Figure 3. *The stride as a trade-off between memory consumption and parallelism of execution. Rectangular boxes represent graph traversal in batches. The pipers involved (N-1, N, N+2) are shown on the right (explanation in text).*

Inter-process communication

A major aspect - and often bottleneck - of parallel computing is inter-process communication (IPC; Fig. 4) [LiYa00]. In PaPy, IPC occurs between parallel pipers connected in a workflow. The communication process is two-stage and involves a manager process - *i.e*, the local Python interpreter used to start the workflow (Fig. 4). A coordinating process is necessary because the connected nodes might evaluate functions in processes with no parent/child relationship. If communication occurs between processes on different hosts, an additional step of IPC (involving a local and a remote RPyC process) is present. Inter-process communication involves data serialization (*i.e.* representation in a form which can be sent or stored), the actual data-transmission (*e.g.* over a network socket) and, finally, de-serialization on the recipient end. Because the local manager process is involved in serializing (de-serializing) data to (from) each parallel process, it can clearly limit pipeline performance if large amounts of data are to be communicated.

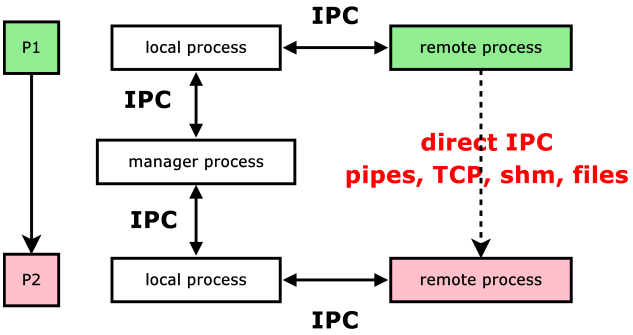


Figure 4. *Inter-process communication (IPC) between pipers (p1, p2). The dashed arrow illustrates possible direct IPC. Communication between the local and remote processes utilizes RPyC (explanation in text).*

PaPy provides functionality for direct communication of producer and consumer processes, thereby mostly eliminating the manager process from IPC and alleviating the bottleneck described above. Multiple serialization and transmission media are supported. In general, the producer makes data available (*e.g.* by serializing it and opening a network socket) and sends only information needed by the consumer end to locate the data (*e.g.* the host and port of the network socket) *via* the manager process. The consumer end receives this information and reads the data. Direct communication comes at the cost of losing platform-independence, as the operating system(s) have to properly support the chosen transmission medium (*e.g.* Unix pipes). Table 2 summarizes PaPy's currently available options.

Table 2: Direct inter-process communication methods.[2]

Method	OS	Remarks
socket	all	Communication between hosts connected by a network.
pipe	UNIX-like	Communication between processes on a single host.
file	all	The storage location needs to be accessible by all processes - *e.g* over NFS or a SAMBA share.
shm	POSIX	Shared memory support is provided by the `posix_shm` library; it is an alternative to communication by pipes.
database	all	Serialized data can be stored as (key, value) pairs in a database. The keys are semi-random. Currently SQLite and MySQL are supported, as provided by mysql-python and sqlite3.

Note that it is possible to avoid some IPC by logically grouping processing steps within a single piper. This is done by constructing a single piper instance from a worker instance created from a tuple of user-written functions, instead of constructing multiple piper instances from single function worker instances. A worker instance is a callable object passed to the constructor of the Piper class. Also, note that any linear, non-branching segment of a pipeline can be collapsed into a single piper. This has the performance advantage that no IPC occurs between functions within a single piper, as they are executed in the same process.

Additional features and notes

Workflow logging

PaPy provides support for detailed workflow logging and is robust to exceptions (errors) within user-written

[2] Currently supported serialization algorithms: pickle, marshall, JSON

functions. These two features have been a major design goal. Robustness is achieved by embedding calls to user functions in a `try ... except` clause. If an exception is raised, it is caught and does not stop the execution of the workflow (rather, it is wrapped and passed as a placeholder). Subsequent pipers ignore and propagate such objects. Logging is supported via the `logging` module from the Python standard library. The `papy` and `IMap` packages emit logging statements at several levels of detail, *i.e.* DEBUG, INFO, ERROR; additionally, a function to easily setup and save or display logs is included. The log is written real-time, and can be used to monitor the execution of a workflow.

Usage notes

A started parallel piper consumes a sequence of N input items (where N is defined by the "stride" argument), and produces a sequence of N resultant items. Pipers are by default "ordered", meaning that an input item and its corresponding result item have the same index in both sequences. The order in which result items become available may differ from the order input items are submitted for parallel processing. In a pipeline, result items of an upstream piper are input items for a downstream piper. The downstream piper can process input items only as fast as result items are produced by the upstream piper. Thus, an inefficency arises if the upstream piper does not return an available result because it is out of order. This results in idle processes, and the problem can be addressed by using a "stride" larger then the number of processes, or by allowing the upstream piper to return results in the order they become available. The first solution results in higher memory consumption, while the second irreversibly abolishes the original order of input data.

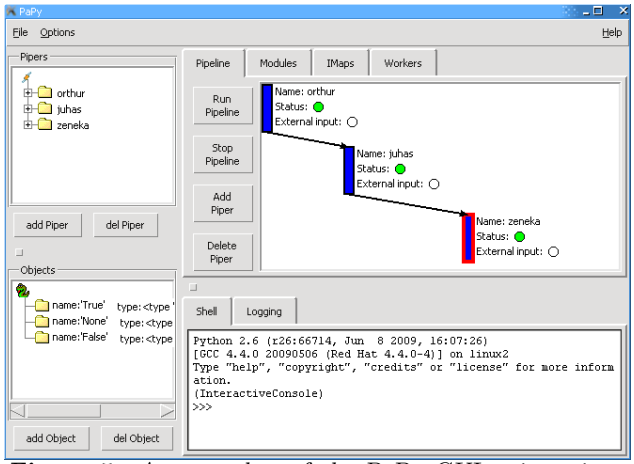


Figure 5. *A screenshot of the PaPy GUI written in Tkinter. Includes an interactive Python console and an intuitive canvas to construct workflows.*

Graphical interface

As a Python package, PaPy's main purpose is to supply and expose an API for the abstraction of a parallel workflow. This has the advantage of flexibility (*e.g.* usage within other Python programs), but requires that the programmer learn the API. A graphical user interface (GUI) is currently being actively developed (Fig. 5). The motivation for this functionality is to allow a user to interactively construct, execute (*e.g.* pause execution), and monitor (*e.g.* view logs) a workflow. While custom functions will still have to be written in Python, the GUI liberates the user from knowing the specifics of the PaPy API; instead, the user explores the construction of PaPy workflows by connecting objects *via* navigation in the GUI.

Workflow construction example

The following code listing illustrates steps in the construction of a distributed PaPy pipeline. The first of the two nodes evaluates a function (which simply determines the host on which it is run), and the second prints the result locally. The first piper is assigned to a virtual resource combining local and remote processes. The scripts take two command line arguments: a definition of the available remote hosts and a switch for using TCP sockets for direct inter-process communication between the pipers. The source code uses the `imports` decorator. This construct allows import statements to be attached to the code of a function. As noted earlier, the imported modules must be available on all hosts on which this function is run.

The pipeline is started, for example, *via*:

```
$ python pipeline.py \
        --workers=HOST1:PORT1#2,HOST2:PORT1#4
```

which uses 2 processes on `HOST1` and 4 on `HOST2`, and all locally-available CPUs. Remote hosts can be started (assuming appropriate firewall settings) by:

```
$ python RPYC_PATH/servers/classic_server.py \
        -m forking -p PORT
```

This starts a RPyC server listening on the specified PORT, which forks whenever a client connects. A forking server is capable of utilizing multiple CPU cores.

The following example (in expanded form) is provided as part of PaPy's online documentation.:

```
#!/usr/bin/env python
# Part 0: import the PaPy infrastructure.
# papy and IMap are separate modules
from papy import Plumber, Piper, Worker
from IMap import IMap, imports
from papy import workers

# Part 1: Define user functions
@imports(['socket', 'os', 'threading'])
def where(inbox):
    result = "input: %s, host:%s, parent %s, \
                    process:%s, thread:%s" % \
    (inbox[0], \
    socket.gethostname(), \
    # the host name as reported by the OS
    os.getppid(), \ # get parent process id
    os.getpid(), \  # get process id
    threading._get_ident())
    # unique python thread identifier
    return result

# Part 2: Define the topology
def pipeline(remote, use_tcp):
    # this creates a IMap instance which uses
    #'remote' hosts.
    imap_ = IMap(worker_num=0, worker_remote=remote)
    # this defines the communication protocol i.e.
    # it creates worker instances with or without
    # explicit load_item functions.
    if not use_tcp:
        w_where = Worker(where)
        w_print = Worker(workers.io.print_)
    else:
        w_where = Worker((where, workers.io.dump_item), \
                         kwargs=({}, {'type':'tcp'}))
        w_print = Worker((workers.io.load_item, \
                         workers.io.print_))
    # the instances are combined into a piper instance
    p_where = Piper(w_where, parallel=imap_)
    p_print = Piper(w_print, debug=True)
    # piper instances are assembled into a workflow
    # (nodes of the graph)
    pipes = Plumber()
    pipes.add_pipe((p_where, p_print))
    return pipes

# Part 3: execute the pipeline
if __name__ == '__main__':
    # get command-line arguments using getopt
    # following part of the code is not PaPy specific
    # and has the purpose of interpreting commandline
    # arguments.
    import sys
    from getopt import getopt
    args = dict(getopt(sys.argv[1:], '', ['use_tcp=', \
                                          'workers='])[0])
    # parse arguments
    use_tcp = eval(args['--use_tcp']) # bool
    remote = args['--workers']
    remote = worker_remote.split(',')
    remote = [hn.split('#') for hn in remote]
    remote = [(h, int(n)) for h, n in remote]
    # create pipeline (see comments in function)
    pipes = pipeline(remote, use_tcp)
    # execution
    # the input to the function is a list of 100
    # integers.
    pipes.start([range(100)])
    # this starts the pipeline execution
    pipes.run()
    # wait until all input items are processed
    pipes.wait()
    # pause and stop (a running pipeline cannot
    # be stopped)
    pipes.pause()
    pipes.stop()
    # print execution statistics
    print pipes.stats
```

Discussion and conclusions

In the context of PaPy, the factors dictating the computational efficiency of a user's pipeline are the nature of the individual functions (nodes, pipers), and the nature of the data linkages between the constituent nodes in the graph (edges, pipes). Although distributed and parallel computing methods are becoming ubiquitous in many scientific domains (*e.g.*, biologically meaningful usec-scale MD simulations [KlLiDrSh09]), data post-processing and analysis are not keeping pace, and will become only increasingly difficult on desktop workstations.

It is expected that the intrinsic flexibility underlying PaPy's design, and its easy resource distribution, could make it a useful component in the scientist's data-reduction toolkit. It should be noted that some data-generation workflows might also be expressible as pipelines. For instance, parallel tempering / replica-exchange MD [EaDe05] and multiple-walker metadynamics [Raiteri06] are examples of intrinsically parallelizable algorithms for exploration and reconstruction of free energy surfaces of sufficient granularity. In those computational contexts, PaPy could be used to orchestrate data *generation* as well as data aggregation / reduction / analysis.

In conclusion, we have designed and implemented PaPy, a workflow-engine for the Python programming language. PaPy's features and capabilities include: (1) construction of arbitrarily complex pipelines; (2) flexible tuning of local and remote parallelism; (3) specification of shared local and remote resources; (4) versatile handling of inter-process communication; and (5) an adjustable laziness/parallelism/memory trade-off. In terms of usability and other strengths, we note that PaPy exhibits (1) robustness to exceptions; (2) graceful support for time-outs; (3) real-time logging functionality; (4) cross-platform interoperability; (5) extensive testing and documentation (a 60+ page manual); and (6) a simple, object-oriented API accompanied by a preliminary version of a GUI.

Availability

PaPy is distributed as an open-source, platform- independent Python (CPython 2.6) module at http://muralab.org/PaPy, where extensive documentation also can be found. It is easily installed *via* the Python Package Index (PyPI) at http://pypi.python.org/pypi/papy/ using `setuptools` by `easy_install papy`.

Acknowledgements

We thank the University of Virginia for start-up funds in support of this research.

References

[TeStYo09] Terwilliger TC, Stuart D, Yokoyama S "Lessons from structural genomics" *Ann.Rev. Biophys.* (**2009**), *38*, 371-83.

[KlLiDr09] Klepeis JL, Lindorff-Larsen K, Dror RO, Shaw DE "Long-timescale molecular dynamics simulations of protein structure and function" *Current Opinions in Structural Biology* (**2009**), *19(2)*, 120-7.

[CoAnCh09] Cock PJ, Antao T, Chang JT, *et al.* "Biopython: freely available Python tools for computational molecular biology and bioinformatics" *Bioinformatics* (**2009**), *25(11)*, 1422-3.

[Knight07] Knight, R et al. "PyCogent: a toolkit for making sense from sequence" *Genome Biology* (**2007**), *8(8)*, R171

[OBHu08] O'Boyle NM, Hutchison GR "Cinfony - combining Open Source cheminformatics toolkits behind a common interface" *Chemistry Central Journal* (**2008**), *2*, 24.

[Hinsen00] Hinsen K "The Molecular Modeling Toolkit: A New Approach to Molecular Simulations" *Journal of Computational Chemistry* (**2000**), *21*, 79-85.

[GrNiLe07] Grünberg R, Nilges M, Leckner J "Biskit--a software platform for structural bioinformatics" *Bioinformatics* (**2007**), *23(6)*, 769-70.

[OiAdFe04] Oinn T, Addis M, Ferris J, *et al.* "Taverna: a tool for the composition and enactment of bioinformatics workflows" *Bioinformatics* (**2004**), *20(17)*, 3045-54.

[ThTaLi05] Thain D, Tannenbaum T, Livny M, "Distributed Computing in Practice: The Condor Experience" *Concurrency and Computation: Practice and Experience* (**2005**), *17*, 2-4, 323-356.

[Ham08] Fiers MW, van der Burgt A, Datema E, de Groot JC, van Ham RC "High-throughput bioinformatics with the Cyrille2 pipeline system" *BMC Bioinformatics* (**2008**), *9*, 96.

[DeGh08] Dean J, Ghemawat S "MapReduce: Simplified Data Processing on Large Clusters" *Comm. of the ACM* (**2008**), *51*, 107-113.

[EaDe05] Earl, D. J. and M. W. Deem "Parallel tempering: Theory, applications, and new perspectives" *Phys. Chem. Chem. Phys.* (**2005**) *7(23)*, 3910-3916.

[Raiteri06] Raiteri P, Laio A, Gervasio FL, Micheletti C, Parrinello M. J *J Phys Chem B.* (**2006**), *110(8)*, 3533-9.

[LiYa00] Liu, P., Yang, C. "Locality-Preserving Dynamic Load Balancing for Data-Parallel Applications on Distributed-Memory Multiprocessors."" (**2000**)

[SkHiMc96] Skillicorn, D. B., Hill, J. M. D. & Mccoll, W. F. "Questions and answers about BSP" (**1996**).

[Knight05] Knight S, "Building Software with SCons," *Computing in Science and Engineering* (**2005**), *7(1)*, 79-88.

[Lloyd94] Lloyd JW, "Practical advantages of declarative programming" (**1994**)

[Sunderam90] Sunderam V.S. "PVM: A framework for parallel distributed computing" *Concurrency: Practice and Experience* (**1990**), 2, *4*, 315-339

[KlLiDrSh09] Klepeis JL, Lindorff-Larsen K, Dror RO, Shaw DE "Long-timescale molecular dynamics simulations of protein structure and function" *Curr. Opin. Struc. Biol.* (**2009**), *19(2)*, 120-7.

PMI - Parallel Method Invocation

Olaf Lenz (lenzo@mpip-mainz.mpg.de) – *Max Planck Institute for Polymer Research, Postfach 3148, D-55021 Mainz* Germany

The Python module "pmi" (Parallel Method Invocation) is presented. It allows users to write simple, non-parallel Python scripts that use functions and classes that are executed in parallel.
The module is well suited to be employed by other modules and packages that want to provide functions that are executed in parallel. The user of such a module does not have to write a parallel script, but can still profit from parallel execution.

Introduction

All modern CPUs provide more than one core, so that they can execute several tasks in parallel. Still, most software, and in particular self-written scripts, do not exploit this capability. The reason for this is mainly, that parallel programming is significantly harder than writing serials programs.

In particular in a scientific environment, one often has to use computationally intensive functions that could greatly benefit from parallelism. Still, scientific packages (like SciPy) usually provide only serial implementations of these functions, as providing a parallel function would mean that the user would have to call it from a parallel program.

When parallelizing a program, one usually has the choice between two fundamental models: the shared-memory thread model and the distributed-memory message-passing model [Bar07] .

In the shared-memory thread model, the different parallel threads have access to shared variables that they can use to exchange information. Programming in the world of threads is relatively simple, however, it is also relatively simple to produce problems like deadlocks and race conditions, or to create highly inefficient code, as all communication between the threads happens implicitly via the shared variables. Furthermore, the model can only be employed efficiently on machines that provide shared memory that can be accessed by all threads.

In the distributed-memory message-passing model, the parallel tasks (or processes) all have an independent data space. To exchange information, the tasks have to use explicit message-passing functions. Writing message-passing programs is more tedious than writing threaded programs, as all communication has to be made explicit. On the other hand, message-passing programs are less error-prone, as each task has its own, independent data space, so that certain kinds of deadlocks and race conditions can not happen. Another advantage of using message-passing parallelization is that it can be used on distributed-memory machines, but that it can also be easily and efficiently mapped to shared-memory platforms, while it is not possible to map threaded programs onto distributed-memory machines.

Therefore, when using a shared-memory machine, the choice between the models is mostly a choice of the programming style, but it does not influence how an algorithm is parallelized. Many algorithms, in particular in scientific computing where large datasets are processed, are algorithms that can profit from data parallelism. In data-parallel programs, the different tasks do not execute completely independent operations, instead each task does the same operation, but on a different piece of data. In particular, most control structures and the program logic is the same in all tasks. This can greatly simplify writing parallel programs [Boy08].

Data-parallel programming can be used in both parallelization models. In the world of C/C++ and FORTRAN, data-parallel programming in the shared-memory thread model is supported by the fork-join programming model of OpenMP [OpenMP] [CDK01] . In this model, the programmer marks regions of the program to be executed in parallel, while the rest of the program is running in a serial fashion. At the beginning of this region, the flow of control is forked, and the code of the region is executed on all threads in parallel. If not marked otherwise, all variables are shared, so that all threads can access the same data, and can work on their piece of data. At the end of the parallel region, all threads are joined back, and it returns to a single flow of control. Using OpenMP is relatively simple and has a few nice properties. In particular, it is easy to synchronize the threads and to implement control structures and the program logic, as these can be implemented in serial fashion, so that only the time-consuming parts have to be parallelized. On the other hand, in the parallel regions, the programmer is exposed to all dangers of threaded programming, like race conditions, deadlocks and inefficient access to the shared memory.

Data-parallel programming in the message-passing model is supported by the standardized library MPI (Message Passing Interface) [MPI] [MPIF09]. Each parallel task runs completely independent, however MPI provides advanced communication operations that allows the tasks to explicitly communicate. Writing data-parallel algorithms in this framework is somewhat more tedious, as it requires a lot of explicit communication, in particular when it comes to implementing the program logic and control structures. On the other hand, it is not so easy to fall into the traps of parallel programming, like producing inefficient code or race conditions.

When writing data-parallel programs, it would be good if one could combine at least some of the advantageous features of both programming models. To this end, it helps to understand that the fork-join model of OpenMP makes it simple to implement control structures and the program logic is independent of the underlying parallel programming model. The notion of parallel regions could be used in both the shared-memory thread model, where the threads have access to shared variables, as well as in the distributed-memory message-passing model, where the different tasks can exchange messages. However, to the author's best knowledge, combining the fork-join model with message-passing for data-parallel programming has not been done so far.

The PMI module

In the course of the ESPResSo++ [ESPResSo] project, the module `pmi` (Parallel Method Invocation) [PMI] has been developed. It is a pure Python module that tries to combine the fork-join model with message-passing. Like this, it allows the user to write the program logic and control structures in the fashion of OpenMP, while it still uses the less error-prone MPI for the actual parallelization. The only requirement of the module is a working MPI module (for example mpi4py [mpi4py] or boostmpi [boostmpi].

PMI does not go so far to allow a programmer to simply mark certain regions of the program to be run in parallel. Instead, it allows users to call - from a serial script - arbitrary Python functions to be executed in parallel (e.g. on multicore CPUs or on large parallel machines). Once called, the different invocations of the function can communicate via MPI. When the function returns, the flow of control is returned to the serial script. Furthermore, PMI allows to create parallel object instances, that have a corresponding instance in all parallel tasks, and to call arbitrary methods in these objects.

PMI has two possible areas of use: on the one hand, it allows modules or packages to provide parallel functions and classes. A user can call these from a simple, apparently serial Python script, that in fact runs parallel code, without the user having to care about parallelization issues. On the other hand, PMI could be used within a GUI that is used to control parallel code.

Other than modules that base on thread parallelization, scripts using PMI and MPI can be used on multicore machines as well as on convenience clusters with fast interconnects and big high-performance parallel machines.

When comparing PMI to the standard multithreading or multiprocessing modules, it must be stressed that PMI has all the advantages that message-passing parallelization has over thread-parallelization: it can work on both shared-memory as well as on distributed memory-machines, and it provides the less error-prone parallelization approach.

When comparing PMI to using the pure MPI modules or other message-passing solutions (like PyPar [PyPar]), it has the advantage that it doesn't require the programmer to write a whole parallel script to use a parallel function. Instead, only those functions that actually can use parallelism have to be parallelized. PMI allows a user to hide the parallelism in those functions that need it.

To the best knowledge of the author, the only other solution that provides functionality comparable to PMI are the parallel computing facilities of IPython [IPython]. Using these, it would be possible to write parallel functions that can be called from a serial script. Note, however, that IPython is a significantly larger package and the parallel facilities have a number of strong dependencies. These dependencies make it hard to run it on some more exotic high-performance platforms like the IBM Blue Gene, and prohibit its use within simple libraries.

Parallel function calls

Within PMI, the task executing the main script is called the *controller*. On the controller, the pmi commands `call()`, `invoke()` or `reduce()` can be called, which will execute the given function on all workers (including the task running the controller itself). The three commands differ only in the way they handle return values of the called parallel functions. Furthermore, the command `exec_()` allows to execute arbitrary Python code on all workers.

In the following example, we provide the outline of the module `mandelbrot_pmi` that contains a function to compute the Mandelbrot fractal in parallel:

```
import pmi
# import the module on all workers
pmi.exec_('import mandelbrot_pmi')

# This is the parallel function that is
# called from mandelbrot()
def mandelbrot_parallel((x1, y1), (x2, y2),
                        (w, h), maxit):
   '''Compute the local slice of the
      mandelbrot fractal in parallel.'''
   # Here we can use any MPI function.
     .
     .
     .

# This is the serial function that can be
# called from a (serial) user script
def mandelbrot(c1, c2, size, maxit):
   return pmi.call(
      'mandelbrot_pmi.mandelbrot_parallel',
      c1, c2, size, maxit)
```

A user can now easily write a serial script that calls the parallelized function `mandelbrot`:

```
import pmi, mandelbrot_pmi
# Setup pmi
pmi.setup()
# Call the parallel function
M = mandelbrot_pmi.mandelbrot(
  (-2.0, -1.0), (1.0, 1.0),
  (300, 200), 127)
```

Parallel class instances

`pmi.create()` will create and return a parallel instance of a class. The methods of the class can be invoked via `call()`, `invoke()` or `reduce()`, and when the parallel instance on the controller is used as an argument to one of these calls, it is automatically translated into the corresponding instance on the worker.
Taking the following definition of the class `Hello` in the module `hello`:

```
from mpi4py import MPI
class Hello(object):
  def __init__(self, name):
    self.name = name
    # get the number of the parallel task
    self.rank = MPI.COMM_WORLD.rank
  def printmsg(self):
    print("Hello %s, I'm task %d!" %
          (self.name, self.rank))
```

Now, one could write the following script that creates a parallel instance of the class and call its method:

```
import pmi
pmi.setup()
pmi.exec_('import hello')
hw = pmi.create('hello.Hello', 'Olaf')
pmi.call(hw, 'printmsg')
```

This in itself is not very useful, but it demonstrates how parallel instances can be created and used.

Parallel class instance proxies

To make it easier to use parallel instances of a class, PMI provides a metaclass `Proxy`, that can be used to create a serial frontend class to a parallel instance of the given class. Using the metaclass, the module `hello_pmi` would be defined as follows:

```
import pmi
from mpi4py import MPI
pmi.exec_('import hello_pmi')

# This is the class to be used in parallel
class HelloParallel(object):
  def __init__(self, name):
    self.name = name
    self.rank = MPI.COMM_WORLD.rank
  def printmsg(self):
    print("Hello %s, I'm task %d!" %
          (self.name, self.rank))

# This is the proxy of the parallel class,
# to be used in the serial script
class Hello(object):
  __metaclass__ = pmi.Proxy
  pmiproxydefs = \
    dict(cls = 'HelloParallel',
         pmicall = [ 'printmsg' ])
```

Given these definitions, the parallel class could be used in a script:

```
import pmi, hello_pmi
pmi.setup()
hello = hello_pmi.Hello('Olaf')
hello.printmsg()
```

Summary

The PMI module provides a way to call arbitrary functions and to invoke methods in parallel. Using it, modules and packages can provide parallelized functions and classes to their users, without requiring the users to write error-prone parallel script code.

References

[PMI] http://www.espresso-pp.de/projects/pmi/

[Bar07] B. Barney, *Introduction to Parallel Computing*, Lawrence Livermore National Laboratory, 2007, http://www.llnl.gov/computing/tutorials/parallel_comp/

[Boy08] C. Boyd, *Data-parallel computing*, ACM New York, NY, USA, 2008, http://doi.acm.org/10.1145/1365490.1365499

[OpenMP] http://openmp.org/wp/

[CDK01] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2001

[MPI] http://www.mcs.anl.gov/research/projects/mpi/

[MPIF09] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 2.2*, High Performance Computing Center Stuttgart, Germany, 2009, http://www.mpi-forum.org/docs/docs.html

[ESPResSo] http://www.espresso-pp.de

[mpi4py] http://mpi4py.scipy.org/

[boostmpi] http://mathema.tician.de/software/boostmpi

[PyPar] M. Cieślik and C. Mura, *PaPy: Parallel and distributed data-processing pipelines in Python*, in Proc. SciPy 2009, G. Varoquaux, S. van der Walt, J. Millman (Eds), pp. 17–24, http://sourceforge.net/projects/pypar/

[IPython] F. Perez and B. Granger: *Ipython, a system for interactive scientific computing*, Computing in Science & Engineering, 9(3), 21-29, 2007 http://ipython.scipy.org/doc/stable/html/parallel/index.html

Sherpa: 1D/2D modeling and fitting in Python

Brian L. Refsdal (brefsdal@head.cfa.harvard.edu) – *Harvard-Smithsonian Center for Astrophysics*, USA
Stephen M. Doe (sdoe@head.cfa.harvard.edu) – *Harvard-Smithsonian Center for Astrophysics*, USA
Dan T. Nguyen (dtn@head.cfa.harvard.edu) – *Harvard-Smithsonian Center for Astrophysics*, USA
Aneta L. Siemiginowska (aneta@head.cfa.harvard.edu) – *Harvard-Smithsonian Center for Astrophysics*, USA
Nina R. Bonaventura (nina@head.cfa.harvard.edu) – *Harvard-Smithsonian Center for Astrophysics*, USA
Douglas Burke (dburke@cfa.harvard.edu) – *Harvard-Smithsonian Center for Astrophysics*, USA
Ian N. Evans (evans_i@head.cfa.harvard.edu) – *Harvard-Smithsonian Center for Astrophysics*, USA
Janet D. Evans (janet@head.cfa.harvard.edu) – *Harvard-Smithsonian Center for Astrophysics*, USA
Antonella Fruscione (antonell@head.cfa.harvard.edu) – *Harvard-Smithsonian Center for Astrophysics*, USA
Elizabeth C. Galle (egalle@head.cfa.harvard.edu) – *Harvard-Smithsonian Center for Astrophysics*, USA
John C. Houck (houck@space.mit.edu) – *MIT Kavli Institute*, USA
Margarita Karovska (karovska@head.cfa.harvard.edu) – *Harvard-Smithsonian Center for Astrophysics*, USA
Nicholas P. Lee (nlee@head.cfa.harvard.edu) – *Harvard-Smithsonian Center for Astrophysics*, USA
Michael A. Nowak (mnowak@space.mit.edu) – *MIT Kavli Institute*, USA

Sherpa is a modern, general purpose fitting and modeling application available in Python. It contains a set of robust optimization methods that are critical to the forward fitting technique used in parametric data modeling. The Python implementation provides a powerful software package that is flexible and extensible with direct access to all internal data objects. Sherpa affords a highly proficient scientific working environment required by the challenges of modern data analysis. It is implemented as a set of Python modules with computationally-intensive portions written in C++/FORTRAN as extension modules using the Python C-API. It also provides a high level user interface with command-like functions in addition to the classes and functions at the API level. Sherpa is being developed by the Chandra X-ray Center (CXC) and is packaged with the Chandra data analysis software package (CIAO). Sherpa can also be built as a standalone application; it can be extended by the user, or embedded in other applications. It allows for analysis specific to astronomy, but also supports generic modeling and fitting tasks. The 'astro' module includes additional astronomy model functions, FITS image support, instrument models, and utilities. Sherpa's model library includes some commonly used 1D and 2D functions and most of the X-ray spectral models found in the High Energy Astrophysics Science Archive Research Center (HEASARC) XSPEC application. Sherpa also supports user-defined models written in Python, C++, and FORTRAN, allowing users to extend Sherpa with models not included in our model library. Sherpa has a set of optimization methods including LMDIF, implementations of Differential Evolution (Monte Carlo) and Nelder-Mead simplex. These functions minimize differences between data points and model values (as measured by a fit statistic such as the chi-squared, maximum likelihood, or a user-defined statistic). The generic I/O module includes back-end interfaces to read ASCII files using NumPy and astronomy image files (FITS) using PyFITS or CIAO Crates (CXC Data Model library in C++). Sherpa is general enough to fit and model data from a variety of astronomical observatories (e.g., Chandra, ROSAT, Hubble) and over many wavebands (e.g., X-ray, optical, radio). In fact, Sherpa can fit and model any data set that can be represented as collections of 1D or 2D arrays (and can be extended for data of higher dimensionality). Data sets can also be simulated with noise using any model. The visualization module also allows for multiple back-ends. An interface to Matplotlib and CIAO ChIPS (CXC plotting package layered on VTK in C++) are provided for line and histogram plotting. 2D visualization is supported by the Smithsonian Astrophysical Observatory (SAO) imager, DS9. The Sherpa command line uses a configured version of IPython to provide a high level shell with IPython 'magic' and readline support.

Introduction

Chandra is one of NASA's great observatories and astronomers from all over the world continue to use it for X-ray astronomy since its launch in 1999. Sherpa is one of many tools included in the Chandra Interactive Analysis of Observations (CIAO) [ciao] software package. Sherpa is a multi-dimensional, robust Python application that handles the task of modeling and fitting in Chandra data analysis. It is developed by a team of programmers and scientists in the Chandra X-ray Center (CXC) and many of the algorithms and numerical methods have been updated and optimized for the latest computing architectures. In past releases, Sherpa was comprised of a rigid YACC parser and much legacy C++ code and recently has been re-implemented into a cleaner, modular form.

Fitting and modeling

Fitting a model to data in Sherpa is done by modifying one or more model parameters until the differences are minimized between the predicted data points and

the raw data. Scientists model data to find values that map to physical quantities, such as temperature, that cannot easily be measured from the data set alone. The analysis is not complete until the model expression contains the appropriate number of parameters, that the parameter values are known to some degree of confidence, and the probability of attaining such a fit by chance is acceptably low.

Use

Sherpa is primarily used for Chandra X-ray data analysis. Many X-ray astronomers have used it for years to analyze their data, and have published the results based on Sherpa fitting and modeling. Sherpa is also used in the data pipeline of the Chandra Source Catalog (CSC) [csc] to estimate source sizes and spectral properties.

Design

To achieve an extensible, general purpose fitting and modeling application, we chose to implement a series of Python modules that we later joined to form the fitting application. At the object layer, many of these modules include C++ extensions for computationally intensive routines.

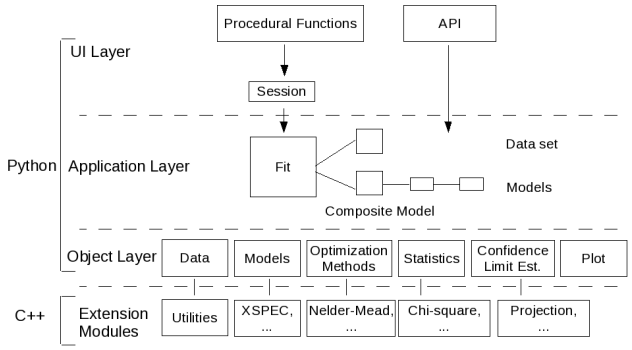


Figure 1. *Sherpa Module Design*

Sherpa can read data from a variety of astronomical observatories (e.g., Chandra, ROSAT, Hubble), as they all publish their data in the Flexible Image Transport System (FITS) format; Sherpa can deal with data in any waveband (X-ray, optical, or radio). Even other scientific or tabular data in ASCII format are supported. Data sets can even be simulated using a defined model and added noise.

Sherpa provides a set of 1D and 2D models, as well as an interface to the XSPEC model library (an X-ray spectral package published by the High Energy Astrophysics Science Archive Research Center (HEASARC) [xspec]. Users can also extend Sherpa by writing models of their own in Python, C++, or FORTRAN. An interface to a Fast Fourier Transform (FFT) library is available for convolutions to model the effects of point spread functions (PSFs), as well as some physical effects (e.g., spectral line broadening).

Sherpa provides robust optimization functions to handle the low signal-to-noise often found in X-ray data. They include Levenberg-Marquardt (LMDIF) [lm] and in-house implementations of Differential Evolution (Monte Carlo) [mc] and Nelder-Mead simplex [nm].

To complement the optimization functions, Sherpa includes native chi-squared and maximum likelihood fit statistics. The interface is generic enough that users can also define their own fit statistic functions.

Confidence limits for fitted parameter values can be calculated with the function, *projection*. To accurately estimate confidence limits for a given parameter, a multidimensional confidence region needs to be projected onto the parameter's associated axis in parameter space. For a given parameter, the projection function searches for the locations on that axis where the confidence region is projected. A new function, *confidence*, is a pure Python implementation that takes the same approach, but is much more efficient in searching parameter space and generally is both more robust and efficient than the projection function. Both methods compute confidence limits for each parameter independently and are currently computed in parallel on multi-core CPUs.

I/O interface

The I/O interface provides a middle layer where multiple back-ends can be used to support multiple file readers. Basic ASCII file reading is available using NumPy. More complicated astronomy file formats, like FITS, are standard. For example, Crates is a CXC FITS reader that supports the Chandra Data Model. Crates is the default I/O back-end and comes bundled with the CIAO software package. Alternatively, PyFITS [pyfits] can be used in a standalone installation of Sherpa. Each back-end (for Crates and PyFITS) interfaces to the same front-end that exposes I/O functions to other parts of Sherpa. Top-level I/O functions such as *load_data()* are written to use that front-end, so the choice of a particular back-end for I/O remains hidden from the rest of the application.

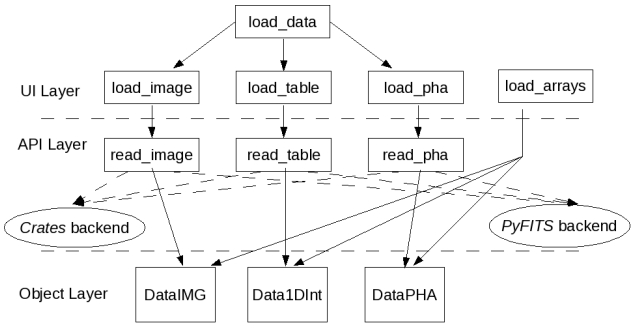


Figure 2. *Sherpa I/O Interface*

Visualization interface

Similarly, the visualization interface supports multiple back-ends for line and contour plotting and 2D imag-

ing. ChIPS is a CIAO package available for line and contour plotting written in C++ on top of VTK.

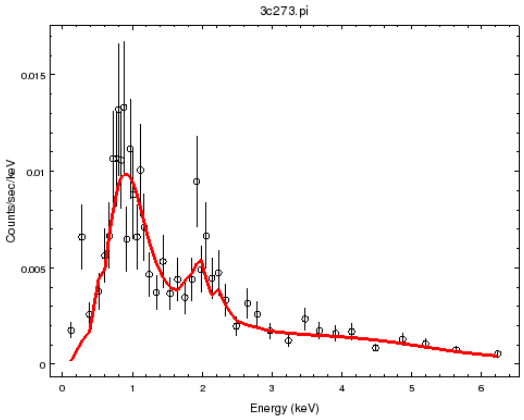


Figure 3. *ChIPS line plot of X-ray data with fitted model*

Sherpa includes a back-end to matplotlib [mpl] as an alternative for standalone installations. DS9 [ds9], the SAO imager, is the primary back-end for image visualization.

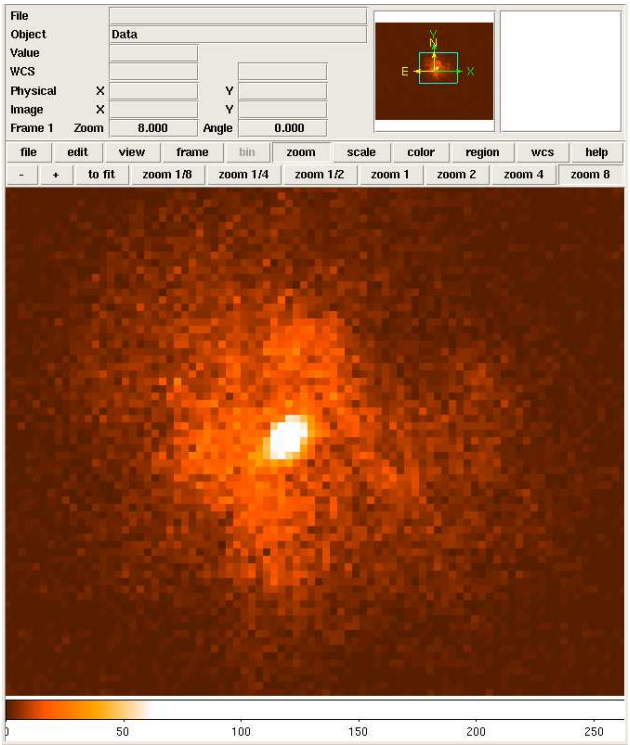


Figure 4. *DS9 image of X-ray source*

With generic interfaces, Sherpa offers users the freedom to add their own back-ends as needed.

API

The Sherpa UI that developers can build upon comes in two levels, the basic API and a set of procedural functions. The API includes all the additional functionality for X-ray analysis, imported like a normal

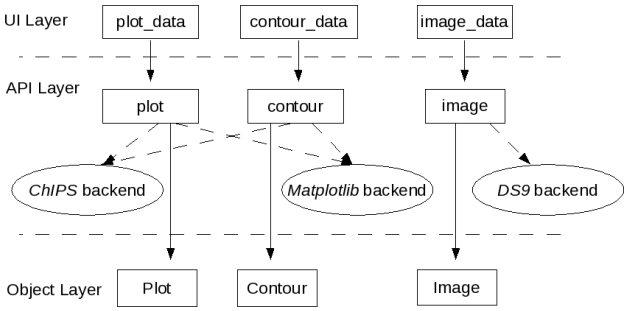


Figure 5. *Sherpa Visualization Interface*

Python package. A typical example of an X-ray spectral fit, such as modeling the redshifted photo-electric absorption of a quasar, is shown below.

Import the base and astronomy modules

```
>>> from sherpa.all import *
>>> from sherpa.astro.all import *
```

Read in a data set from file and setup a filter from 0.3-7.5 keV.

```
>>> pha = read_pha('q1127_src1.pi')
>>> pha.notice(0.3, 7.5)
```

Instantiate individual model classes and setup initial parameter values, freezing and thawing parameters as necessary.

```
>>> abs1 = XSphabs('abs1')
>>> abs1.nH = 0.041
>>> abs1.nH.freeze()
>>> zabs1 = XSzphabs('zabs1')
>>> zabs1.redshift=0.312
>>> p1 = PowLaw1D('p1')
```

Inspection of the source data set can provide clues to the initial parameter values. Some simple Sherpa models include functions to estimate the initial parameter values, based on the data. The algorithms for such "guess" functions are basic (e.g., maximum value, average, determining full-width half max) and do not necessarily perform well for composite models.

```
>>> p1.guess(*pha.to_guess(), limits=True, values=True)
```

A scripting language like Python allows users to define their composite models at run time. Here, the composite model is defined as a product of the three and convolved with the instrument response.

```
>>> model = standard_fold(pha, abs1*zabs1*p1)
```

The Fit object is initialized with class instances of data, model, statistic, optimization method, and confidence limit method.

```
>>> f = Fit(pha, model, Chi2DataVar(), LevMar(),
            Projection())
>>> results = f.fit()
```

The fit results object includes the fitted parameter values and the additional information calculated during the fit. They include the initial, final, and change in statistic values, the number of function evaluations

used by the optimization method, the number of data points, the degrees of freedom, the null hypothesis probability (Q-value), and the reduced statistic value.

```
>>> print(results.format())
Method                = levmar
Statistic             = chi2datavar
Initial fit statistic = 17917.4
Final fit statistic   = 686.013 at function evaluation 22
Data points           = 494
Degrees of freedom    = 491
Probability [Q-value] = 1.27275e-08
Reduced statistic     = 1.39717
Change in statistic   = 17231.4
   zabs1.nH       0.094812
   p1.gamma       1.28615
   p1.ampl        0.000705228
```

In determining if the model suitably represents the data, the maximum likelihood ratio (MLR) can be computed. Given two models, the MLR can determine which model better describes a particular data set. The more complex model should be picked when the ratio is less than 0.05. Once a fit has been run and the model selected that best describes the data set, Sherpa can estimate the confidence limits using the projection algorithm.

Estimate 1 sigma confidence limits using projection. The projection results object displays the upper and lower parameter bounds with the best-fit values.

```
>>> results = f.est_errors()
>>> print(results.format())
Confidence Method     = projection
Fitting Method        = levmar
Statistic             = chi2datavar
projection 1-sigma (68.2689%) bounds:
   Param            Best-Fit  Lower Bound  Upper Bound
   -----            --------  -----------  -----------
   zabs1.nH         0.094812  -0.00432843   0.00436733
   p1.gamma          1.28615  -0.00968215   0.00970885
   p1.ampl       0.000705228 -6.63203e-06  6.68319e-06
```

Procedural UI

The same script can be run in a more compact form using the procedural UI. Users can perform common operations --such as reading files, defining models, and fitting --by calling predefined fuctions, without having to write their own code in Python.

Import all astronomy procedural functions

```
>>> from sherpa.astro.ui import *
```

Read data set from file and setup a filter

```
>>> load_data('q1127_src1.pi')
>>> notice(0.3, 7.5)
```

Model instantiation uses a unique syntax of the form 'modeltype.identifier' to create a model and label it in a single line. xsphabs is a model class with abs1 as the identifier and the expression xsphabs.abs1 returns an instance of the class with rich comparison methods. Model parameter values are initialized similarly to the API.

```
>>> set_model(xsphabs.abs1*xszphabs.zabs1*powlaw1d.p1)
>>> abs1.nH = 0.041
>>> freeze(abs1.nH)
>>> zabs1.redshift=0.312
>>> guess(p1)
```

The statistic can be set as an instance of a Sherpa statistic class or a string identifier to native Sherpa statistics.

```
>>> set_stat('chi2datavar')
```

Execute the fit and display the results using a logger.

```
>>> fit()
...<fit output>
```

Compute the confidence limits and display the results using a logger.

```
>>> proj()
...<confidence limit output>
```

Confidence

Projection estimates the confidence limits for each parameter independently (currently this can be done in parallel). Projection searches for the N-sigma limits, where N-sigma corresponds to a given change in the fit statistic from the best-fit value.

For the chi-squared fit statistic, the relation between sigma and chi-squared is $\sigma = \sqrt{\Delta\chi^2}$. For our maximum likelihood fit statistics, the relation has the form $\sigma = \sqrt{2 * \Delta \log L}$. Projection has the added feature that if a new minimum is found in the boundary search, the process will restart using the new found best-fit values. The accuracy of the confidence limits using the projection and confidence methods is based on the assumption that the parameter space is quadratic, where in, the fit statistic function for a given parameter can be expanded using a Taylor series. Also, Sherpa assumes that the best-fit point is sufficiently far ($\approx 3\sigma$) from the parameter space boundaries. Cases where these assumptions do not hold, users should use an alternative approach such as Markov Chain Monte Carlo (MCMC) to map the parameter space using specific criteria. MCMC support within Sherpa is currently in research and development and should be available in future releases.

Fit Statistics

Sherpa has a number of χ^2 statistics with different variances. The χ^2 fit statistic is represented as

$$\chi^2 \equiv \sum_i \frac{(D_i - M_i)^2}{\sigma_i^2},$$

where D_i represents the observed data, M_i represents the predicted model counts, and σ_i^2 represents the variance of the sampling distribution for D_i.

Sherpa defines many flavors of the χ^2 statistic with different variances. The native flavors of the variance include leastsq

$$\sigma^2 \equiv 1,$$

chi2constvar

$$\sigma^2 \equiv \frac{\sum_{i=1}^{N} D_i}{N},$$

chi2modvar

$$\sigma^2 \equiv M_i,$$

chi2gehrels

$$\sigma^2 \equiv [1 + \sqrt{D_i + 0.75}]^2,$$

chi2datavar

$$\sigma^2 \equiv D_i \qquad \text{for } D_i > 0,$$

and chi2xspecvar

$$\sigma^2 \equiv \begin{cases} D_i & \text{if } D_i > 0 \\ 1 & \text{if } D_i = 0 \end{cases}.$$

Sherpa likelihood statistics include Cash

$$C \equiv 2 \sum_i [M_i - D_i \log M_i] \propto -2 \log L,$$

and the C-statistic

$$C \equiv 2 \sum_i [M_i - D_i + D_i(\log D_i - \log M_i)],$$

where D_i represents the observed data, M_i represents the predicted model counts, and L, the log-likelihood

$$L \equiv \prod_i \frac{M_i^{D_i}}{D_i!} \exp(-M_i).$$

Visualization

Visualizing the parameter space can help determine if the assumptions of projection hold or not. 1D line plots of the statistic against parameter values are available with the interval projection function. For a given parameter, the function steps away from the best-fit value, and refits to find the statistic value at a number of points in parameter space. Then a curve showing how the statistic varies with parameter value is drawn.

```
>>> int_proj(p1.gamma)
```

2D confidence contours can be drawn with the region projection function. For any two parameters, a grid is constructed, such that the function refits at each point to find the fit statistic. This maps out parameter space around the best-fit parameter values.
Confidence contours can then be drawn (corresponding to $1\sigma, 2\sigma, 3\sigma$ confidence by default. The plot boundaries are set to be 4σ by default (assuming that parameter space boundaries are no closer than $\approx 4\sigma$ from the best-fit values).

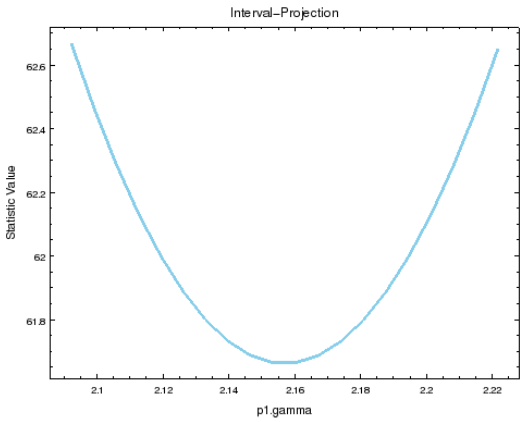


Figure 6. *Interval projection line plots typically look parabolic in a well behaved parameter space.*

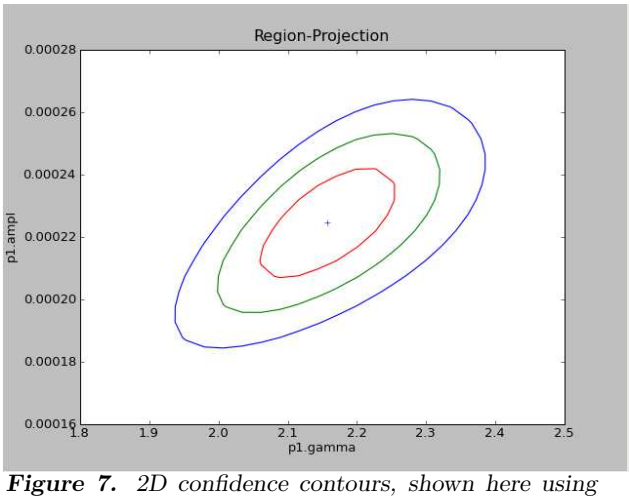


Figure 7. *2D confidence contours, shown here using matplotlib, are a typical product of a Sherpa session. Contours provide the parameter value expectation to some degree of confidence. The smaller the contour, the more constrained the best-fit values are.*

```
>>> reg_proj(p1.gamma, p1.ampl)
```

Once parameter space has been mapped out, then the line or contour plots can provide users with a visual of parameter space around the best-fit parameter values. For 1D line and contour plotting the Sherpa high-level UI includes many convenience functions that hide much of the API boiler-plate in the plot module.

```
>>> plot_fit_delchi()
```

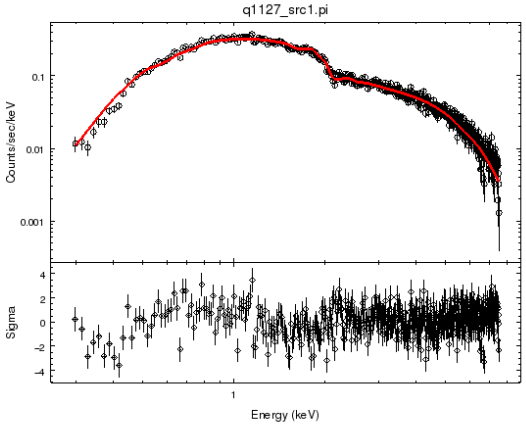


Figure 8. *ChIPS line plot of source data set and error bars, fitted model, and delta chi-squared residuals*

Convenience functions are also available for 2D imaging using an interface to DS9 with the XPA messaging system [xpa].

```
>>> image_fit()
```

User Models

Users who wish to use their own models in Sherpa can follow the user model interface. User defined models can be written in Python, C++ or FORTRAN. An example of a user-defined model in Python using the Sherpa high-level UI is shown below.

```
>>> def calc(pars, x, **kwargs):
    """
    y = m*x + b
    """
    return pars[0]*x + b

>>> load_user_model(calc, "mymodel")
>>> add_user_pars("mymodel", ["m", "b"], [-3, 5])
```

In the function signature for calc, pars is a list of user defined parameter values and x is a NumPy array representing the model's grid.

Conclusion

Sherpa is a mature, robust fitting and modeling application, with continuing development. The Python code is modular and extensible with plug-in back-ends, and is flexible enough for general use.

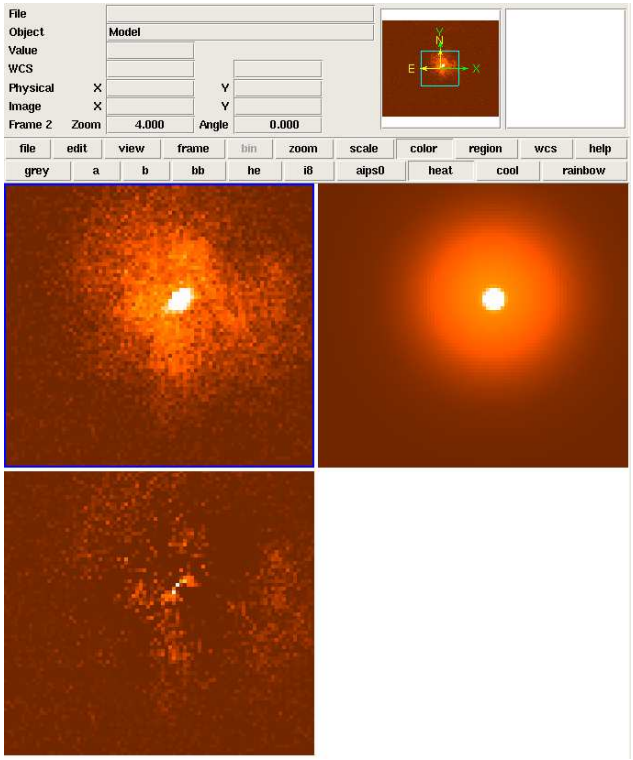


Figure 9. *DS9 displays source image with fitted model and residuals.*

Users can download a source tarball and install Sherpa standalone [alone] or download it with the many tools included in the CIAO software package [bundle]. Documentation is available with Python help, CIAO ahelp files, and web pages that detail each function [ahelp] and that show scripting threads [threads].

Users creating standalone installations using distutils are required to install dependencies like Python, NumPy, and Fastest Fourier Transform in the West (FFTW) [fftw].

Future plans for Sherpa include an implementation of MCMC for complicated parameter spaces (provided by the CHASC astro-statistics group [chasc]); speed and performance improvements using parallel techniques for model evaluation; improved documentation (possibly using Sphinx [sphinx]); and a web-based version control system that allows users to download the latest stable version of Sherpa.

Support of the development of Sherpa is provided by National Aeronautics and Space Administration through the Chandra X-ray Center, which is operated by the Smithsonian Astrophysical Observatory for and on behalf of the National Aeronautics and Space Administration contract NAS8-03060.

References

[ahelp] http://cxc.harvard.edu/sherpa/ahelp/index_python.html

[alone] http://hea-www.harvard.edu/uinfra/sherpa/Documentation/download/index.html

[bundle] http://cxc.harvard.edu/ciao/download

[chasc] http://hea-www.harvard.edu/AstroStat/

[ciao] http://cxc.harvard.edu/ciao

[csc] http://cxc.harvard.edu/csc

[ds9] http://hea-www.harvard.edu/RD/ds9/

[fftw] M. Frigo and S.G. Johnson, *The Design and Implementation of FFTW3*, Proceedings of the IEEE 93 (2), 216–231 (2005). Special Issue on Program Generation, Optimization, and Platform Adaptation. http://www.fftw.org/

[lm] Lecture Notes in Mathematics 630: Numerical Analysis, G.A. Watson (Ed.), Springer-Verlag: Berlin, 1978, pp. 105-116

[mc] R. Storn, and K. Price, *Differential Evolution: A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces*, J. Global Optimization 11, 1997, pp. 341-359 http://www.icsi.berkeley.edu/~storn/code.html

[mpl] J.D. Hunter, *Matplotlib: A 2D graphics environment.* Computing in Science and Engineering. 9: 90-95 (2007). http://matplotlib.sourceforge.net.

[nm] J.A. Nelder and R. Mead, Computer Journal, 1965, vol 7, pp. 308-313. Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, Paul E. Wright *Convergence Properties of the Nelder-Mead Simplex Algorithm in Low Dimensions*, SIAM Journal on Optimization, Vol. 9, No. 1 (1998), pp. 112-147. http://citeseer.ist.psu.edu/3996.html. Wright, M. H. (1996) *Direct Search Methods: Once Scorned, Now Respectable* in Numerical Analysis 1995 (Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis) (D.F. Griffiths and G.A. Watson, eds.), 191-208, Addison Wesley Longman, Harlow, United Kingdom. http://citeseer.ist.psu.edu/155516.html

[pyfits] http://www.stsci.edu/resources/software_hardware/pyfits

[sphinx] G. Brandl, *Sphinx, Python documentation generator*, http://sphinx.pocoo.org/

[threads] http://cxc.harvard.edu/sherpa/threads/all.html

[xpa] http://hea-www.harvard.edu/saord/xpa/

[xspec] http://heasarc.gsfc.nasa.gov/docs/xanadu/xspec/

The FEMhub Project and Classroom Teaching of Numerical Methods

Pavel Solin (solin@unr.edu) – *University of Nevada, Reno,,* USA
Ondrej Certik (ondrej@certik.cz) – *University of Nevada, Reno,,* USA
Sameer Regmi (sregmi@unr.edu) – *University of Nevada, Reno,,* USA

We introduce briefly the open source project FEMhub and focus on describing how it can be used for live demonstrations of elementary numerical methods in daily classroom teaching.

The FEMhub Project

FEMhub [femhub] is an open source distribution of finite element (FEM) codes with unified Python interface, developed by the *hp*-FEM group at the University of Nevada, Reno [hpfemorg]. The aim of FEMhub is to establish common standards in the development of open source FEM codes, allow for accuracy and performance comparisons of different codes, and provide a common platform for collaboration and exchange of modules.
Currently, FEMhub contains the open source codes FiPy, Hermes [hermes], Phaml and SfePy as FEM engines, tools to ease visualisation (matplotlib [mpl], mayavi [mayavi], pyglet [pgl]), standard Python libraries Scipy [scipy], Numpy [numpy] and Sympy, and a web notebook which is based on the Sage notebook.

Interactive Web Notebook

The goal of the FEMhub web notebook [femhub-nb] is to make all FEM codes in FEMhub available remotely through any web browser. Inside the web notebook, one will be able to define geometry, generate a mesh, specify boundary and initial conditions, define arbitrary partial differential equations (PDE) to be solved, package the components and send them for processing to a remote high-performance computing facility (currently UNR Research Grid), and visualize the results once they are received.

Teaching Numerical Methods

We have found students' understanding of the fundamentals of numerical methods to be an obstacle to learning *hp*-FEM algorithms efficiently. As a result, we decided to use the web notebook to implement a series of elementary numerical methods that the students can work with interactively and thus gain a much better understanding of the material. The notebook does not employ the CPU of the machine where it is executed, and therefore one can use it to compute on desktop PCs, laptops, netbooks and even iphones. In particular, one can use it in every classroom that has Internet access.
The response of the students was very positive, therefore we started to add new worksheets systematically and to utilize the notebook in the classroom regularly. We found that by running the methods in real time, we can get much more across about their strengths, weaknesses and typical behavior than ever before. Last but not least, the students stopped grumbling about programming homework assignments.
Through this paper, we would like to share our positive experience with anyone who teaches elementary numerical methods. All worksheets described below are freely available at http://nb.femhub.org. Right now (as of November 2009) you still need to create an account to access them, but we are working currently on eliminating this and making the notebook even more open.

Taylor Polynomial

The Taylor polynomial *T(x)* is an approximation to a function *f(x)* in the vicinity of a given point *a*, based on the knowledge of the function value *f(a)*, first derivative *f'(a)*, second derivative *f"(a)*, etc. In the web notebook, one has two ways of defining *T(x)*: Via the point *a*, list *[f(a), f'(a), f"(a), ...]*, and the endpoints:

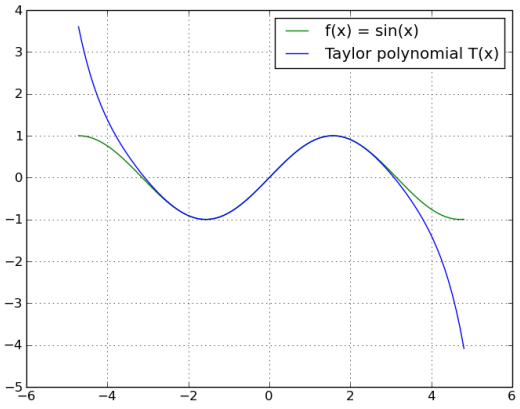
```
taylor_1(0, [0, 1, 0, -1, 0, 1, 0, -1], -3*pi/2, 3*pi/2)
```

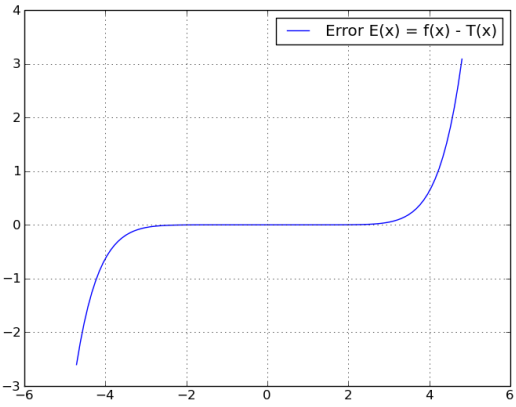
or by entering the point *a*, the function *f(x)*, *x* for independent variable, degree *n*, and the endpoints:

```
taylor_2(0, sin(x), x, 7, -3*pi/2, 3*pi/2).
```

Both these commands generate the same *T(x)* but the latter also plots the error:

```
f(x) = sin(x)
T(x) = x - x**3/6 + x**5/120 - x**7/5040
```





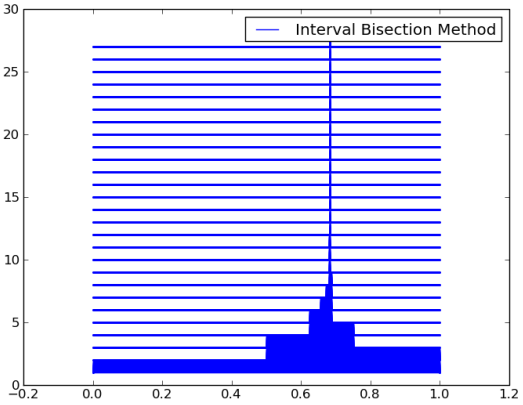
Using the notebook, one can demonstrate obvious facts such as that with increasing n the polynomial $T(x)$ gets closer to $f(x)$ in the vicinity of the point a, but also that the textbook recipe may overestimate the error $|f(x) - T(x)|$ substantially, that $T(x)$ usually diverges from $f(x)$ quickly once $|x-a|$ exceeds some threshold, that $T(x)$ has a special form at $a=0$ if $f(x)$ is even or odd, etc. See the Taylor Polynomial worksheet at [femhub-nb] for more details.

Rootfinding Techniques

The three most-widely used methods (interval bisection, Newton's method, and fixed-point iteration) are called as one would expect:

```
bisection(1/(1+x*x) - x, x, 0, 1, 1e-8)
newton(1/(1+x**2) - x, x, 1, 1e-8)
fixed_point(1/(1+x**2), x, 1, 1e-8)
```

For the bisection method, we like to illustrate the textbook formula for the number of steps needed to obtain the root with a given accuracy. For the Newton's method we show how extremely fast it usually is compared to the other two techniques but also that it can fail miserably when the initial guess is far from the root. For the fixed-point iteration we like to show how slow it typically is compared to the other two methods, but also that it may work in situations where the Newton's does not. And of course, that it can fail if the derivative of the underlying function exceeds the interval $(-1,1)$.



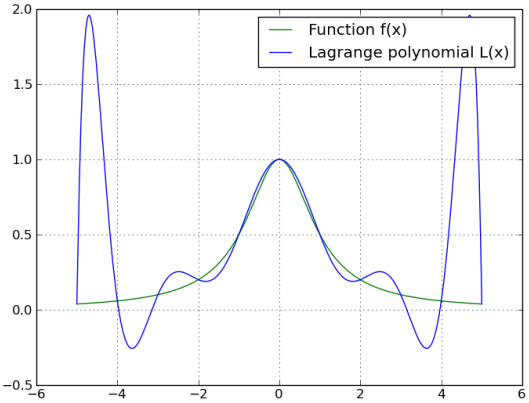
The previous image illustrates the history of interval subdivisions produced by the bisection method.

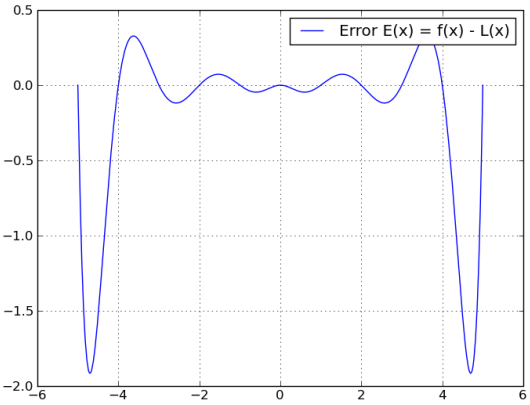
Lagrange Interpolation Polynomial

The Lagrange interpolation polynomial $L(x)$ is a single polynomial of degree n passing through $n+1$ given points of the form [x, y] with distinct x-coordinates. The worksheet offers four different problem types:

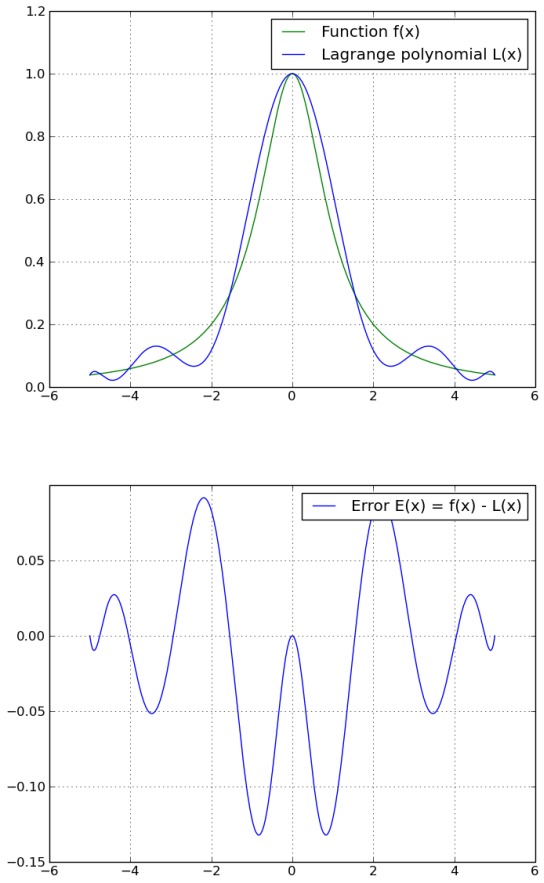
1. Given is an arbitrary array of x-coordinates and an arbitrary array of y-coordinates.
2. Given is an arbitrary array of x-coordinates and a function $f(x)$ for y-coordinates.
3. Given is an array of equally-spaced x-coordinates and a function $f(x)$ for y-coordinates.
4. Given are Chebyshev points for x-coordinates and a function $f(x)$ for y-coordinates.

We like to use option #1 to show the elementary interpolation functions that are equal to one at one of the points and zero at the remaining ones. Option #2 is useful for showing the difference between the Lagrange and Taylor polynomials. Options #3 and #4 can be used to demonstrate the notoriously bad performance of equally-spaced interpolation points and that one should use the Chebyshev points instead. The following two figures illustrate interpolation of the famous function $1/(1+x^2)$ in the interval (-5, 5) on 11 equidistant points and the huge error that one ends up with:





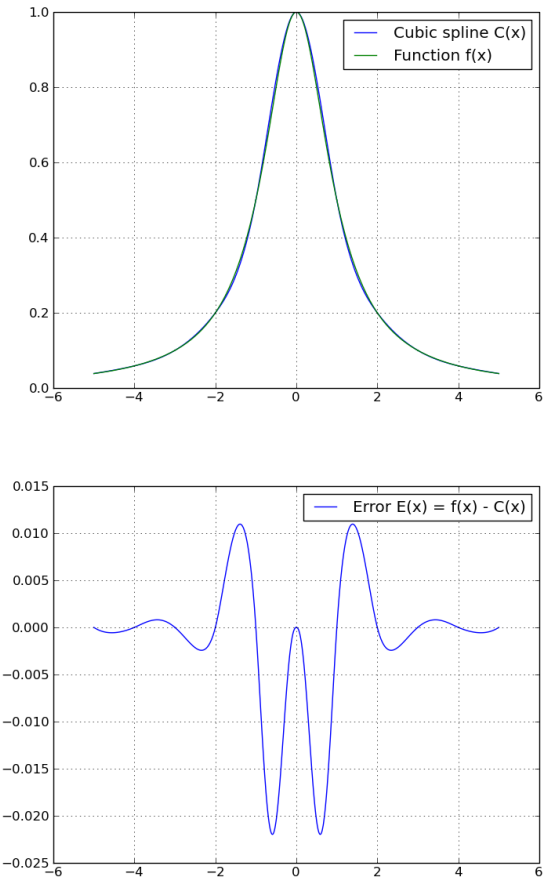
The following pair of figures shows the same situation, only the equidistant points are replaced with Chebyshev points. The reader can see that the error *E(x) = f(x) - L(x)* drops about 13 times in magnitude:



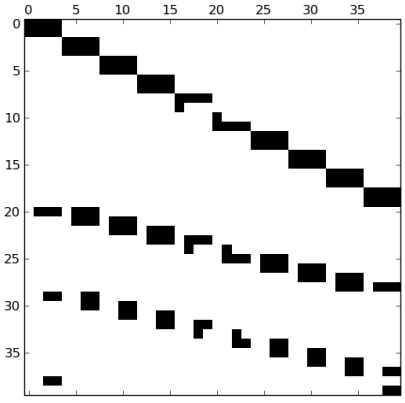
In option #4, one can demonstrate nicely the optimality of the Chebyshev points by moving slightly some of them to the left or right (in the part of the code where they are actually defined) and observing that the interpolation error always goes up. See the Lagrange Polynomial worksheet at [femhub-nb] for more details.

Cubic Splines

Interpolation via cubic splines is more popular than the Lagrange interpolation because the results do not contain wiggles and look much more natural. The basic setting is the same as in Lagrange interpolation - one has *n+1* points of the form *[x,y]* with distinct *x*-coordinates. These points divide the interval into *n* subintervals. In each of them, we seek a cubic polynomial via four unknown coefficients. This means that we have *4n* unknowns. The corresponding *4n* equations are obtained by requiring the splines to match the end point values in every subinterval (*2n* equations), requiring the first derivatives from the left and right to be the same at all interior points (*n-1* equations), and the same for second derivatives (another *n-1* equations). At this point, two conditions are left to be specified and there is some freedom in them. For example, one can require the second derivatives to vanish at interval endpoints, which results into the so-called *natural cubic spline*. But one can also set the first derivatives (slopes) at interval endpoints, etc. The following pair of images illustrates the interpolation of the function *1/(1+x*x)* in the interval *(-5, 5)* on 11 equidistant points as above, but now with cubic splines. Compared to the Chebyshev interpolant, the error drops 6 times in magnitude.



The following figure shows the sparsity structure of the *4n* times *4n* matrix (zeros in white, nonzeros in black). We like to highlight the underlying matrix problems because in our experience the students usually do not know that matrices can be used outside of a linear algebra course.



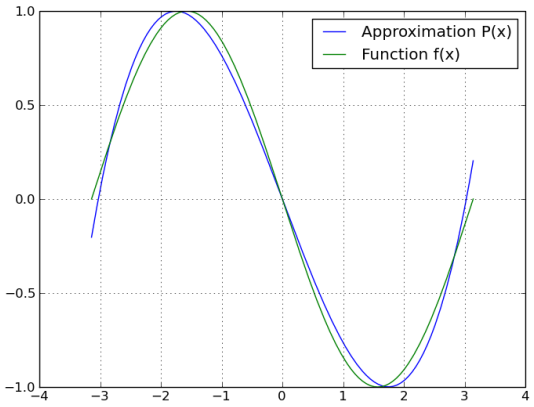
See the Cubic Splines worksheet at [femhub-nb] for more details.

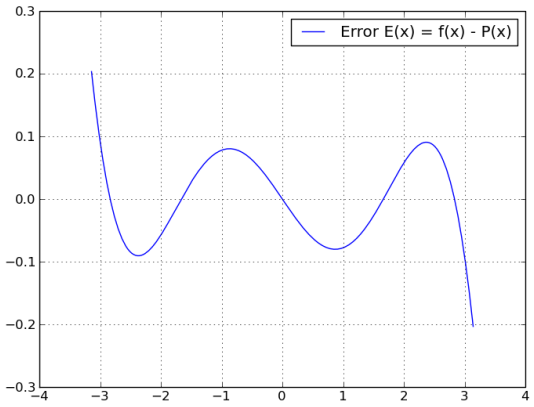
Least-Squares Approximation

The least-squares approximation technique is very different from interpolation - it finds in a given set of polynomials an element that is *closest to the approximated function f(x)*. (There is nothing about matching the function values of *f(x)* exactly.) Typically, the set of polynomials is chosen to be the Legendre polynomials because of their orthogonality, and the distance between is measured in the so-called *L2-norm*. The following commands are used to define a function *f(x)* and calculate its least-squares polynomial approximation of degree *n* in an interval (a,b):

```
# Define function f(x)
def f(x): return -sin(x)

# Calculate and plot in interval (a, b) the
# least-squares polynomial approximation P(x) of f(x)
least_squares(-pi, pi, f, x, 3)
```

The output for these parameters looks as follows:

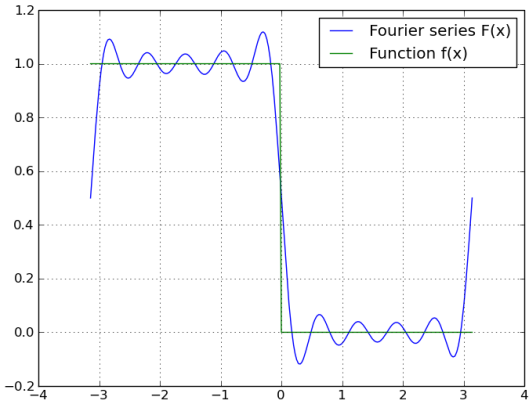




This worksheet also plots the underlying basis functions (Legendre polynomials). One can use elementary integration functions to show the students that indeed the Legendre polynomials are orthogonal in the *L2*-product. We also like to use this opportunity to explain the importance of numerical quadrature by projecting a complicated function that cannot be integrated analytically. For more details see the Least Squares worksheet at [femhub-nb].

Fourier Expansion

This is an excellent opportunity to show the students that the Fourier expansion is nothing else than the least-squares approximation. One just replaces the Legendre polynomials with the functions *1, cos(x), sin(x), cos(2x), sin(2x), ...*, and considers the periodicity interval *(-pi, pi)* instead of a general interval *(a, b)*. Otherwise everything is the same. It is useful to demonstrate to the students that the Fourier basis above indeed is orthogonal in the *L2*-product. The following figure shows the approximation of a piecewise-constant discontinuous signal. The worksheet also plots the error as usual, not shown here for space limitations.



See the Fourier Expansion worksheet at [femhub-nb] for more details.

References

[femhub] http://femhub.org/.

[femhub-nb] http://nb.femhub.org/.

[hermes] http://hpfem.org/main/hermes.php.

[hpfemorg] http://hpfem.org/.

[mpl] J.D. Hunter (2007). *Matplotlib: A 2D graphics environment*. Computing in Science and Engineering. 9: 90-95. http://matplotlib.sourceforge.net/.

[mayavi] P. Ramachandran, G. Varoquaux, *Mayavi: Making 3D Data Visualization Reusable*, in Proceedings of the 7th Python in Science conference (SciPy 2008) http://code.enthought.com/projects/mayavi/.

[numpy] T. Oliphant et al., *NumPy*, http://numpy.scipy.org/.

[pgl] http://www.pyglet.org/.

[scipy] E. Jones, T. Oliphant, P. Peterson, *SciPy: Open source scientific tools for Python* http://www.scipy.org/.

Exploring the future of bioinformatics data sharing and mining with Pygr and Worldbase

Christopher Lee (leec@chem.ucla.edu) – *Department of Chemistry Biochemistry, UCLA, 611 Charles Young Dr. East, Los Angeles, CA 90095* USA
Alexander Alekseyenko (alexander.alekseyenko@nyumc.org) – *Center for Health Informatics and Bioinformatics, Department of Medicine, New York University School of Medicine, New York, NY 10016* USA
C. Titus Brown (ctb@msu.edu) – *Dept. of Computer Science and Engineering, Dept. of Microbiology and Molecular Genetics, Michigan State University, East Lansing, Michigan 48824* USA

Worldbase is a virtual namespace for scientific data sharing that can be accessed via "from pygr import worldbase". Worldbase enables users to access, save and share complex datasets as easily as simply giving a specific name for a commonly-used dataset (e.g. Bio.Seq.Genome.HUMAN.hg17 for draft 17 of the human genome). Worldbase transparently takes care of all issues of how to access the dataset, what code must be imported to use it, what dependencies on other datasets it may have, and how to make use of its relations with other datasets as specified by its schema. Worldbase works with a wide variety of "back-end" storage, including data stored on local file systems, relational databases such as MySQL, remote services via XMLRPC, and "downloadable" resources that can be obtained from the network but automatically installed locally by Worldbase.

Introduction

One of the most important challenges in bioinformatics is a pure Computer Science problem: making it easy for different research groups to access, mine and integrate each other's datasets, in ways that go beyond simple web browsing. The problems of bioinformatics data sharing have grown enormously as the scale and complexity of datasets have increased. Even expert bioinformaticians often find it difficult to work with datasets from outside their specialization; non-expert users often find it impossible. Such difficulties are one of the main barriers to delivering the full value of genomics to all researchers who could benefit from it. Even when good analysis tools are available, the difficulties of accessing and integrating large datasets often limit who can do interesting analyses [LI01].

It doesn't have to be this way. Enabling datasets to "flow" automatically from one place to another, to inter-connect through cross-references that work automatically, and to always bring along the code modules necessary for working with them - these are all infrastructural principles that computer scientists have solved for other domains [DK76]. Good infrastructure should be transparent. Like the Domain Name Service and other technologies that power the web, it should just work, without being visible to the user.

The need for both computational and human scalability

One major obstacle to easy sharing of bioinformatics datasets is their sheer scale and complex interdependencies. Some aspects of this are very simple; for example, many prospective users just don't have enough disk space on their computer to load their desired data. In that case, they should be able to access the data over network protocols transparently; that is, with the exact same interfaces used if the data were stored locally. But a deeper problem is the fact that existing systems for accessing data from other scientists rely on *expertise*: "if the user is an expert in how to use this particular kind of data, s/he'll figure out how to find, download, install, and use this dataset". Since most experts are inexpert at most things, this approach does not scale [BITL]. We have therefore developed a simple but general data-sharing mechanism called `worldbase` [Pygr]:

- To use any public dataset, all the user needs to know is its *name*.
- Asking for a dataset by name yields a full Python interface to it, enabling the user to mine it in all the ways that the original producer of the data could.
- All datasets should be accessible by name from any networked computer, using the closest available resources, or if the user requests, by automatically downloading and installing it locally. Just like the World Wide Web, `worldbase` seeks to create a facade of fully integrated data, not just from the user's computer or LAN but from the whole world.
- The interface for using that data should be exactly the same no matter how or where it is actually being accessed from.
- To use a dataset's relationships to other datasets, again all the user needs to know is the *name* of one of its attributes that links it to another dataset. If the user requests such an attribute, the linked dataset(s) will again be accessed automatically.

In this paper we first describe the current version of `worldbase` (Pygr v.0.8.0, Sept. 2009) by illustrating typical ways of using it. We then discuss some principles of scalable data integration that we believe will prove to be general across many domains of scientific computing.

Using Worldbase

Retrieving a dataset from Worldbase

Say you want to work with human genome draft 18. Start Python and type the following:

```
>>> from pygr import worldbase
>>> hg18 = worldbase.Bio.Seq.Genome.HUMAN.hg18()
```

That's it: you now have the human genome dataset, and can start working. Let's see how many contigs it contains, pull out one chromosome, and print a sequence interval of interest, using standard Python methods:

```
>>> len(hg18)
49
>>> chr1 = hg18['chr1']
>>> len(chr1)
247249719
>>> ival = chr1[20000000:20000500]
>>> print ival
cctcggcctcccaaagtgctgggattacaggcgtgagccaccgcgcagcc...
```

`worldbase` establishes a one-step model for accessing data: *ask for it by name*:

- `worldbase` is an importable namespace for all the world's scientific datasets. You simply *import* the namespace (in the usual Python way), and ask it to *construct* an instance of the dataset that you want (in the usual Python way).
- Of course, this is a *virtual* namespace - you don't actually have all the world's datasets sitting on your computer in a file called `worldbase.py`! `worldbase` connects to a wide variety of data sources (some of which may be on your computer, and some which may be on the Internet) to find out the set of available resources, and then serves them to you.
- `worldbase` takes advantage of Pygr's scalable design. Pygr is first and foremost a *a system of representation* not tied to any fixed assumptions about storage. Pygr is built around the notion of delayed and incremental execution, whereby pieces of a complex and large dataset are loaded only as needed, and in an automatic way that makes this interface transparent. For example, `chr1` represents human chromosome 1, but does not necessarily mean that the human chromosome 1 sequence (245 Mb) was loaded into a Python object). Thus it can work very naturally with huge datasets, even over a network connection where the actual data is stored on a remote server. In this case, `worldbase` is accessing hg18 from UCLA's data server, which is included by default in `worldbase` searches (of course, you can change that).
- To get a dataset, all you need to know is its *name* in `worldbase`. Note that we did not even have to know what code is required to work with that data, let alone explicitly import those modules. `worldbase` takes care of that for you.
- The call syntax (`hg18()`) emphasizes that this acts like a Python constructor: it constructs a Python object for us (in this case, the desired `seqdb.SequenceFileDB` object representing this genome database).
- Note that we did *not* even have to know how to construct the hg18 object, e.g. what Python class to use (`seqdb.SequenceFileDB`), or even to import the necessary modules for constructing it.
- Where did this data actually come from? Since your computer presumably does not contain a local copy of this dataset, `worldbase` accessed it from UCLA's public `worldbase` server over XMLRPC. Currently, our server provides over 460 standard datasets for comparative genomics (such as whole genome sequences, and multigenome alignments), both for client-server access and automated download and installation of datasets (see below).

Storing data in Worldbase

`worldbase` saves not just a data file but a complete Python interface to a dataset, i.e. the capability to *use* and mine the data in whatever ways are possible programmatically. One way of thinking about worldbase is that retrieving data from it is like returning to the moment in time when those data were originally saved to worldbase. Anything you could do with the original data, you can do with the retrieved data.
There are only a few requirements:

- you have your dataset loaded in Python as an object. When retrieved from worldbase, this dataset will be usable by the exact same interface as the original object.
- your object must be **picklable**. Worldbase can store any object that is compatible with standard Python pickling methods. Thus, worldbase is not restricted to Pygr data - but most Pygr classes are of course designed to be stored in worldbase.
- your object must have a docstring, i.e. a `__doc__` attribute. This should give a simple explanatory description so people can understand what this dataset is.

For example, say we want to add the hg17 (release 17 of the human genome sequence) as "Bio.Seq.Genome.HUMAN.hg17" (the choice of name is arbitrary, but it's best to choose a good convention and follow it consistently):

```
from pygr import seqdb, worldbase
# Open the human genome sequence
hg17 = seqdb.SequenceFileDB('hg17')
# Documentation is required to store in worldbase
hg17.__doc__ = 'human genome sequence draft 17'
# Store in worldbase as this name
worldbase.Bio.Seq.Genome.HUMAN.hg17 = hg17
worldbase.commit()
```

Note that you *must* call the function `worldbase.commit()` to complete the transaction and save all pending data resources (i.e. all those added since your last `worldbase.commit()` or `worldbase.rollback()`). In particular, if you have added data to worldbase during a given Python interpreter session, you should always call `worldbase.commit()` or `worldbase.rollback()` prior to exiting from that session.
In any subsequent Python session, we can now access it directly by its worldbase name:

```
from pygr import worldbase
hg17 = worldbase.Bio.Seq.Genome.HUMAN.hg17()
```

You should think of `worldbase` not as a conventional *database* (a container for storing a large set of a specific kind of data) but rather as a *metadata database*, i.e. a container for storing *metadata* describing various datasets (which are typically stored in other, standard databases). By "metadata" we mean information about the *content* of a particular dataset (this is what allows `worldbase` to reload it automatically for the user, without the user having to know what classes to import or how to construct the object correctly), and about its *relations* with other datasets. Throughout this paper, we will use the term "metabase" to refer to this concept of a "metadata database". Whereas a *database* actually stores an entire dataset, a *metabase* merely stores a small amount of metadata pointing to that database and describing its relations with other datasets.

Worldbase automatically captures dataset dependencies

What if you wanted to save a dataset that in turn requires many other datasets? For example, a multigenome alignment dataset is only useful if you also have the genome datasets that it aligns. `worldbase` is smart about figuring out data resource dependencies. For example, you could just save a 17-genome alignment in a single step as follows:

```
from pygr import cnestedlist, worldbase
nlmsa = cnestedlist.NLMSA('/loaner/ucsc17')
nlmsa.__doc__ = 'UCSC 17way multiz alignment, on hg17'
worldbase.Bio.MSA.UCSC.hg17_multiz17way = nlmsa
worldbase.commit()
```

This works, even though using this 17-genome alignment (behind the scenes) involves accessing 17 `seqdb.SequenceFileDB` sequence databases (one for each of the genomes in the alignment). Because the alignment object (NLMSA) references the 17 `seqdb.SequenceFileDB` databases, worldbase automatically saves information about how to access them too.
However, it would be a lot smarter to give those databases worldbase resource names too:

```
from pygr import cnestedlist, worldbase
nlmsa = cnestedlist.NLMSA('/loaner/ucsc17')
for resID, genome in nlmsa.seqDict.prefixDict.items():
  # 1st save the genomes
  genome.__doc__ = 'genome sequence ' + resID
  worldbase.add_resource('Bio.Seq.Genome.' + resID,
                         genome)
nlmsa.__doc__ = 'UCSC 17way multiz alignment, on hg17'
# now save the alignment
worldbase.MSA.Bio.UCSC.hg17_multiz17way = nlmsa
worldbase.commit()
```

This has several advantages. First, we can now access other genome databases using worldbase too:

```
from pygr import worldbase
# get the mouse genome
mm7 = worldbase.Bio.Seq.Genome.mm7()
```

But more importantly, when we try to load the ucsc17 alignment on another machine, if the genome databases are not in the same directory as on our original machine, the first method above would fail, whereas in the second approach worldbase now will automatically figure out how to load each of the genomes on that machine.

Worldbase schema automatically connects datasets for you

One major advantage of `worldbase` is that it explicitly captures and automatically applies schema information about relationships and interconnections between different datasets. By "schema" we mean the precise connections between two or more collections of data. Such inter-relations are vital for understanding and mining scientific datasets. For example "a genome has genes, and genes have exons", or "an exon is connected to another exon by a splice". Let's say we have two databases from `worldbase`, exons and splices, representing exons in a genome, and splices that connect them, and a mapping splicegraph that stores the many-to-many connections between exons (via splices). We can add splicegraph to `worldbase` and more importantly save its schema information:

```
splicegraph.__doc__ = 'graph of exon:splice:exon links'
worldbase.Bio.Genomics.ASAP2.hg17.splicegraph = splicegraph
worldbase.schema.Bio.Genomics.ASAP2.hg17.splicegraph = \
  metabase.ManyToManyRelation(exons, exons, splices,
                bindAttrs=('next','previous','exons'))
worldbase.commit()
```

This tells `worldbase` that splicegraph is a many-to-many mapping from the exons database onto itself, with "edge" information about each such mapping stored in the `splices` database. Concretely, this means that for each `exon1` to `exon2` connection `splice`, then `splicegraph[exon1][exon2]=splice`. Furthermore, the `bindAttrs` option says that we wish to bind this schema as named attributes to all items in those databases. Concretely, for any object `exon1` from the `exons` database, it makes `exon1.next` equivalent to `splicegraph[exon1]`. That means a user can find all the exons that `exon1` splices to, by simply typing:


```
for exon2,splice in exon1.next.items():
   do something...
```

Note how much this simplifies the user's task. The user doesn't even need to know about (or understand) the `splicegraph` database, nor indeed do anything to load `splicegraph` or its dependency splices. Consistent with the general philosophy of `worldbase`, to use all this, the user only needs to know the name of the relevant attribute (i.e. exons have a `next` attribute that shows you their splicing to downstream exons). Because `worldbase` knows the explicit schema of `splicegraph`, it can automatically load splicegraph and correctly apply it, whenever the user attempts to access the `next` attribute. Note also that neither splicegraph nor splices will actually be loaded (nor will the Python module(s) need to construct them be loaded), unless the user specifically accesses the `next` attribute.

Controlling where Worldbase searches and saves data

`worldbase` checks the environment variable `WORLDBASEPATH` for a list of locations to search; but if it's not set, worldbase defaults to the following path:

```
~,.,http://biodb2.bioinformatics.ucla.edu:5000
```

which specifies three locations to be searched (in order): your home directory; your current directory; the UCLA public XMLRPC server. Worldbase currently supports three ways of storing metabases: in a Python shelve file stored on-disk; in a MySQL database table (this is used for any metabase path that begins with "mysql:"); in an XMLRPC server (this is used for any metabase path that begins with "http:").

Worldbase can install datasets for you locally

What if you want to make `worldbase` download the data locally, so that you could perform heavy-duty analysis on them? The examples above all accessed the data via a client-server (XMLRPC) connection, without downloading all the data to our computer. But if you want the data downloaded to your computer, all you have to do is add the flag `download=True`. For example, to download and install the entire yeast genome in one step:

```
yeast = \
  worldbase.Bio.Seq.Genome.YEAST.sacCer1(download=True)
```

We can start using it right away, because `worldbase` automates several steps:

- `worldbase` first checked your local resource lists to see if this resource was available locally. Failing that, it obtained the resource from the remote server, which basically tells it how to download the data.
- `worldbase` unpickled the `Bio.Seq.Genome.YEAST.sacCer1` `seqdb.SequenceFileDB` object, which in turn requested the `Bio.Seq.Genome.YEAST.sacCer1.fasta` text file (again with `download=True`).
- this is a general principle. If you request a resource with `download=True`, and it in turn depends on other resources, they will also be requested with `download=True`. I.e. they will each either be obtained locally, or downloaded automatically. So if you requested the 44 genome alignment dataset, this could result in up to 45 downloads (the alignment itself plus the 44 genome sequence datasets).
- the compressed file was downloaded and unzipped.
- the `seqdb.SequenceFileDB` object initialized itself, building its indexes on disk.
- `worldbase` then saved this local resource to your local worldbase index (on disk), so that when you request this resource in the future, it will simply use the local resource instead of either accessing it over a network (the slow client-server model) or downloading it over again.

Some scalability principles of data Integration

We and many others have used `worldbase` heavily since its release (in Pygr 0.7, Sept. 2007). For some examples of large-scale analyses based on `worldbase` and Pygr, see [AKL07] [Kim07] [ALS08].
We have found `worldbase` to be a work pattern that scales easily, because it enables any number of people, anywhere, to use with zero effort a dataset constructed by an expert via a one-time effort. Furthermore, it provides an easy way (using `worldbase` schema bindings) to integrate many such datasets each contributed by different individuals; these schema relations can be contributed by yet other individuals. In the same way that many software distribution and dependency systems (such as CRAN or fink) package code, `worldbase` manages data and their schema. Attempts to encapsulate data into a global distribution system are made in CRAN [CRAN], but limited to example datasets for demonstrating the packaged code. Unlike CRAN, `worldbase`'s focus is primarily on data sharing and integration.
Based on these experiences we identify several principles that we believe are generally important for scalable data integration and sharing in scientific computing:

- **Encapsulated persistence:** we coin this term to mean that saving or retrieving a dataset requires nothing more than its *name* in a virtual namespace. Moreover, its dependencies should be saved / retrieved automatically by the same principle, i.e. simply by using their *names*. This simple principle

makes possible innumerable additional layers of automation, because they can all rely on obtaining a given dataset simply requesting its name.

- **Interface means representation, not storage:** the public interface for working with `worldbase` datasets must provide a powerful representation of its data types, that can work transparently with many different back-end storage protocols. For example, the back-end might be an XMLRPC client-server connection across a network, or a MySQL database, or specially indexed files on local disk. By using this transparent interface, user application code will work with any back-end; moreover, users should be able to request the kind of back-end scalability they want trivially (e.g. by specifying `download=True`).

- **Scalability is paramount:** in all of our work, scalability has been a primary driving concern - both computational scalability and human scalability. Scientists will only mine massive datasets using a high-level language like Python (whose high-level capabilities made `worldbase` possible) when this retains high performance. Fortunately, elegant solutions such as Pyrex [Pyrex] and Cython [Cython] make this entirely feasible, by enabling those components that are truly crucial for performance to be coded in optimized C.

- **Metabases are the glue:** we coin the term "metabase" to mean a *metadata database*. `worldbase` is *not* intended to be a database in which you actually store data. Instead it is only intended to store *metadata* about data that is stored elsewhere (in disk files; in SQL databases; in network servers, etc.). Broadly speaking these metadata for each resource include: what kind of data it is; how to access it; its relations with other data (schema and dependencies). Metabases are the heart of `worldbase`.

- **Provide multiple back-ends for 3 standard scalability patterns:** Users most often face three different types of scalability needs: **I/O-bound**, data should be worked with in-memory to the extent possible; **memory-bound**, data should be kept on-disk to the extent possible; **CPU-bound** or **disk-space bound**, data should be accessed remotely via a client-server protocol, as there is either no benefit or no space for storing them locally. As an example, `worldbase` has been used so far mainly with Pygr, a framework of bioinformatics interfaces and back-ends. For each of its application categories (e.g. sequences; alignments; annotations), Pygr provides all three kinds of back-ends, with identical interfaces.

- **Standard data models: containers and mappings**. In Pygr and `worldbase`, data divide into two categories: *containers* (a dictionary interface whose keys are identifiers and whose values are the data objects) and *mappings* that map items from one container (dataset) onto another. In particular, Pygr (the Python Graph database framework) generalizes from simple Python mappings (which store a one-to-one relation) to graphs (many-to-many relations). By following completely standard Python interfaces [Python] for containers, mappings and graphs (and again providing three different kinds of back-ends for each, to cover all the usual scalability patterns), Pygr makes `worldbase`'s simple schema and dependency capabilities quite general and powerful. For example, since Pygr's mapping classes support Python `__invert__()` [Python], `worldbase` can automatically bind schema relations both forwards and backwards.

- **Schema is explicit and dynamic:** We have defined *schema* as the metadata that describe the connections between different datasets. When schema information is not available as data that can be searched, transmitted, and analyzed at run-time, programmers are forced either to hard-wire schema assumptions into their code, or write complex rules for attempting to guess the schema of data at run-time. These are one-way tickets to Coding Hell. `worldbase` is able to solve many data-sharing problems automatically, because it stores and uses the schema as a dynamic graph structure. For example, new schema relations can be added between existing datasets at any time, simply by adding new mappings.

- **The web of data interconnects transparently:** These schema bindings make it possible for data to *appear* to interconnect as one seamless "virtual database" (in which all relevant connections are available simply by asking for the named attributes that connect to them), when in reality the datasets are stored separately, accessed via many different protocols, and only retrieved when user code specifically requests one of these linked attributes. This can give us the best of both worlds: an interface that looks transparent, built on top of back-ends that are modular. In fact, one could argue that this principle is the programmatic interface analogue of the hyperlink principle that drove the success of the hypertext web: a facade of completely inter-connected data, thanks to a transparent interface to independent back-ends. From this point of view one might consider `worldbase` to be a logical extension of the "semantic web" [BHL01], but reoriented towards the scalability challenges of massive scientific computing datasets. We think this transparent interconnection of data is gradually becoming a general principle in scientific computing. For example, the Comprehensive R Archive Network like `worldbase` provides a uniform environment for accessing packages of code + data that can be installed with automatic links to their dependencies, albeit with a very different interface reflecting the limitations of its language environment (R instead of Python).

Current limitations and plan for future development

- Currently, `worldbase` supplies no mechanism for global "name resolution" analogous to the DNS. Instead, the user simply designates a list of `worldbase` servers to query via the `WORLDBASEPATH` environment variable; if not specified it defaults to include the main (UCLA) `worldbase` server. This simple approach lets users easily control where they will access data from; for example, a user will typically give higher precedence to local data sources, so that `worldbase` requests will be obtained locally if possible (rather than from a remote server). This lack of centralization is both a vice and a virtue. On the one hand users are free to develop and utilize resources that best suit their research needs. On the other hand, such lack of centralization may often result in duplication of effort, where two research would work on the same data transformation without knowing of each other's efforts. We believe that these problems should be solved by a centralized mechanism similar to the DNS, i.e. that enables data producers to publish data within "domains" in the name space that they "own", and transparently resolves name requests to the "closest" location that provides it.
- Since `worldbase` uses Python pickling, the well-known security concerns about Python unpickling also apply to `worldbase`. These must be resolved prior to expanding `worldbase` from a user-supplied "access list" to a DNS-like global service. We believe that in a public setting, pickle data should be authenticated by secure digital signatures and networks of trust using widely deployed standards such as GnuPG [GnuPG].

Future developments:

- Support for novel and emerging data types, for example:
  - Genome-wide association study (GWAS) data.
  - Next-generation sequencing datasets, such as RNA-seq, allele specific variation, ChIP-seq, and microbiomic diversity data.
- Increased support for using `worldbase` within common cluster computing systems. This seems like a natural way of being able to seamlessly scale up an analysis from initial prototyping to large-scale cluster computations (very different environments where often data resources cannot be accessed in exactly the same way), by pushing all data access issues into a highly modular solution such as `worldbase`.
- Optimized graph queries.
- Data visualization techniques.
- Ability to push code objects along with the data, so that class hierarchy and appropriate access methods may be installed on the fly. In the context of digitally signed code and networks of trust, this could greatly increase the convenience and ease with which scientists can explore common public datasets.

Acknowledgments

We wish to thank the Pygr Development team, including Marek Szuba, Namshin Kim, Istvan Alberts, Jenny Qian and others, as well as the many valuable contributions of the Pygr user community. We are grateful to the SciPy09 organizers, and to the Google Summer of Code, which has supported 3 summer students working on the Pygr project. This work was also supported from grants from the National Institutes of Health (U54 RR021813; SIB training grant GM008185 from NIGMS), and the Department of Energy (DE-FC02-02ER63421).

References

[LI01] C. Lee, K. Irizarry, *The GeneMine system for genome/proteome annotation and collaborative data-mining. IBM Systems Journal* 40: 592-603, 2001.

[DK76] F. Deremer and H.H. Kron, *Programming In the Large Versus Programming In the Small. IEEE Transactions On Software Engineering*, 2(2), pp. 80-86, June 1976.

[BITL] D.S. Parker, M.M. Gorlick, C. Lee, *Evolving from Bioinformatics in the Small to Bioinformatics in the Large. OMICS*, 7, 34-48, 2003.

[Pygr] The Pygr Consortium, *Pygr: the Python Graph Database Framework*, 2009, http://pygr.org.

[AKL07] A.V. Alekseyenko, N. Kim, C.J. Lee, *Global analysis of exon creation vs. loss, and the role of alternative splicing, in 17 vertebrate genomes. RNA* 13:661-670, 2007.

[Kim07] N. Kim, A.V. Alekseyenko, M. Roy, C.J. Lee, *The ASAP II database: analysis and comparative genomics of alternative splicing in 15 animal species. Nucl. Acids Res.* 35: D93-D98, 2007.

[ALS08] A.V. Alekseyenko, C.J. Lee, M.A. Suchard, *Wagner and Dollo: a stochastic duet by composing two parsimonious solos. Systematic Biology* 57: 772-784, 2008.

[CRAN] *The Comprehensive R Archive Network*, from http://cran.r-project.org/.

[Pyrex] G. Ewing, *Pyrex - a Language for Writing Python Extension Modules.* http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex.

[Cython] S. Behnel, R. Bradshaw, D.S. Seljebotn, *Cython: C Extensions for Python.* http://www.cython.org.

[Python] G. van Rossum and F.L. Drake, Jr., *Python Tutorial*, 2009, http://www.python.org.

[BHL01] T. Berners-Lee, J. Hendler, O. Lassila, *The Semantic Web. Sci. Am.* May 2001.

[GnuPG] *GNU Privacy Guard*, http://www.gnupg.org/

Nitime: time-series analysis for neuroimaging data

Ariel Rokem (arokem@berkeley.edu) – *University of California, Berkeley, Berkeley, CA* USA
Michael Trumpis (mtrumpis@berkeley.edu) – *University of California, Berkeley, Berkeley, CA* USA
Fernando Pérez (Fernando.Perez@berkeley.edu) – *University of California, Berkeley, Berkeley, CA* USA

Nitime is a library for the analysis of time-series developed as part of the Nipy project, an effort to build open-source libraries for neuroimaging research. While nitime is developed primarily with neuroimaging data in mind (espespecially functional Magnetic Resonance Imaging data), its design is generic enough that it should be useful to other fields with experimental time-series. The package starts from a purely functional set of algorithms for time-series analysis, including spectral transforms, event-related analysis and coherency. An object-oriented layer is separated into lightweight data container objects for the representation of time-series data and high-level analyzer objects that couple data storage and algorithms. Each analyzer is designed to deal with a particular family of analysis methods and exposes a high-level object oriented interface to the underlying numerical algorithms. We briefly describe functional neuroimaging and some of the unique considerations applicable to time-series analysis of data acquired using these techniques, and provide examples of using nitime to analyze both synthetic data and real-world neuroimaging time-series.

Introduction

`Nitime` (http://nipy.sourceforge.net/nitime) is a library for time-series analysis of data from neuroimaging experiments, with a design generic enough that it should be useful for a wide wide array of tasks involving experimental time-series data from any source.

`Nitime` is one of the components of the NiPy [NiPy] project, an effort to develop a set of open-source libraries for the analysis and visualization of data from neuroimaging experiments.

Functional MRI: imaging brain activity

One of the major goals of neuroscience is to understand the correspondence between human behavior and activity occurring in the brain. For centuries, physicians and scientists have been able to identify brain areas participating in various cognitive functions, by observing the behavioral effects of damage to those areas. Within the last $\sim$ 25 years, imaging technology has advanced enough to permit the observation of brain activity *in-vivo*. Among these methods, collectively known as functional imaging, fMRI (functional Magnetic Resonance Imaging) has gained popularity due to its combination of low invasiveness, relatively high spatial resolution with whole brain acquisition, and the development of sophisticated experimental analysis approaches. fMRI measures changes in the concentration of oxygenated blood in different locations in the brain, denoted as the BOLD (Blood Oxygenation Level Dependent) signal [Huettel04]. The cellular processes occurring when neural impulses are transmitted between nerve cells require energy derived from reactions where oxygen participates as a metabolite, thus the delivery of oxygen to particular locations in the brain follows neural activity in that location. This fact is used to infer neural activity in a region of the brain from measurements of the BOLD signal therein. In a typical fMRI experiment this measurement is repeated many times, providing a spatio-temporal profile of the BOLD signal inside the subject's brain. The minimal spatial unit of measurement is a volumetric pixel, or "voxel", typically of the order of a few mm^3.

The temporal profile of the acquired BOLD signal is limited by the fact that blood flow into neurally active tissue is a much slower process than the changes in the activity of neurons. From a signal processing perspective, this measured profile can be seen as the convolution of a rapid oscillation at rates of $\mathcal{O}(10-1000)$Hz (neuronal activity) with a much slower function that varies at rates of ~ 0.15Hz (changes in blood flow due to neighboring blood vessels dilating and contracting). This slowly varying blood flow profile is known as the *hemodynamic response function* (HRF), and it acts as a low-pass filter on the underlying brain activity [Aguirre97].

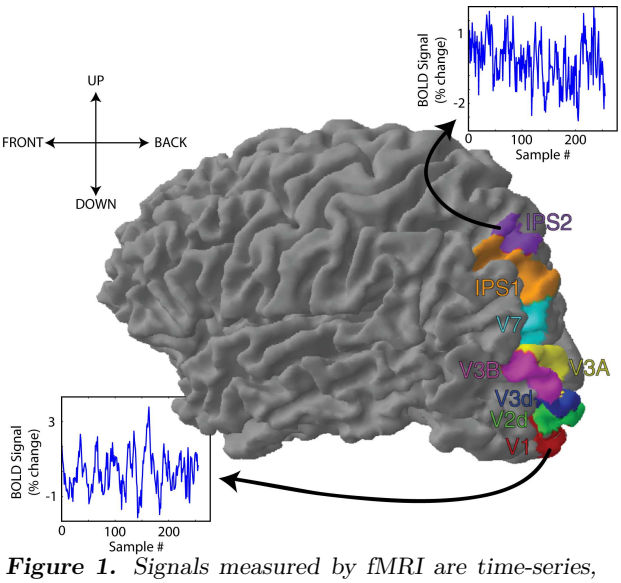


Figure 1. *Signals measured by fMRI are time-series, illustrated from areas in the human visual cortex (adapted from Silver et al., 2005; with permission).*

fMRI data analysis and time-series analysis

The interpretation of fMRI data usually focuses on the analysis of the relationship between the BOLD time-series in each voxel and the occurrence of events of behavioral significance, such as the presentation of a stimulus to the subject or the production of a motor response by the subject.

Figure 1 presents a rendered image of the brain, presented with the anterior part of the brain turning leftwards. Visual areas of the cerebral cortex are located in the posterior part of the brain. The colored patches represent different functional regions in the visual cortex: these areas labeled V1-V7 respond to visual stimulation. Each contains a representation of the visual field such that adjacent locations on the brain surface respond to stimuli appearing in adjacent locations in the visual field. The areas IPS1 and IPS2 (named after their anatomical location in the intraparietal sulcus) contain an ordered representation of the visual field, but respond to the allocation of attention instead of direct visual stimulation [Silver05]. By averaging the measured BOLD signal over all the voxels in each area, we obtain time-series representative of the activity in the region; this is illustrated in the insets for V1 and IPS2.

Typically, univariate analyses calculate a statistical measure of the correlation between the actual activity in each voxel and the activity expected, if the voxel contains neurons which respond to the behaviorally relevant events. Clusters of voxels that significantly correlate with this model are then considered to contain neurons participating in the cognitive function associated with the behavior in question. Sometimes, this approach is sufficient in order to answer interesting questions about functional specialization of various parts of the brain. However, in most cases it is beneficial to further study the time-series measured in the fMRI with approaches that go beyond univariate analysis.

One important aspect in further analysis of the time-series is the definition of regions of interest (ROI) and the targeting of the analysis to those regions [Poldrack06]. ROIs are defined based on criteria such as the known functional specialization of a region in the brain or the anatomical location of that region. In an ROI-based analysis, the time-series from all the voxels in this ROI are extracted and averaged and the average time-series is then subject to further analysis. This approach is readily adopted in areas of the brain where functionally distinct areas can be spatially defined. For example, the areas of the cortex which participate in processing of visual information (often referred to as "visual areas" and denoted by V1, for "primary visual cortex", V2, for "secondary visual cortex", etc.) each contain an ordered representation of the entire visual field (Figure 1) [Wandell07]. These neighboring regions can thus be spatially separated based on the layout of their respective visual field "maps" relative to each other.

One extension of the univariate approach mentioned above, is to examine the functional relations between different time-series, extracted from different locations in the brain. One of the major advantages of fMRI is that measurements are performed simultaneously from many different regions of the brain. Therefore, this method allows us to define not only the correspondence between the time-series of BOLD in one particular region and the events that occurred in the environment while this data was collected, but also the correspondence between time-series collected from one region and the time-series simultaneously collected in another region of the brain. This approach is often referred to as "functional connectivity analysis" [Friston94], where bivariate and multivariate measures of covariance between two or more time-series, taken from different areas in the brain, are calculated. This kind of analysis allows us to infer not only about the participation of a particular brain region in a cognitive process of interest, but also about the functional network of regions and the interplay between activity in these different regions. These analysis techniques can be done using an ROI based approach (see example, below). However, they can also be utilized as exploratory data analysis techniques, in which the connectivity between every pair of voxels in the brain is calculated and the voxels are clustered into functional modules according to their connectivity.

`Nitime`

The `nitime` package tries to provide scientists conducting brain-imaging research with a clean and easy-to-use interface to algorithms that calculate quantities derived from the time-series acquired in fMRI experiments. It contains implementations both of methods previously used in neuroimaging and of time-series analysis techniques that have yet to be widely used in neuroimaging. We also aim to develop new and original ways of analyzing fMRI data and to make these methods available to the community of scientists conducting brain-imaging research through `nitime`. The algorithms are built to allow calculation of some univariate quantities pertaining to the time-series in question, as well as bivariate and multivariate quantities. They are meant to allow an ROI based approach, as well as analyses done at the single-voxel level. Many of the algorithms could be used in the analysis of other kinds of time-series, whether taken from other modalities of neuroscientific research or even in other fields of science. Therefore, we have decoupled the parts of the implementation that are directly related to neuroimaging data (functions for reading data from standard neuroimaging formats, for example), from our object-oriented design and from the implementation of algorithms.

`Nitime` fits within the broader context of other related Python projects: The `TimeSeries` SciPy scikit [TimeSeries] focuses on the analysis of calendar-based time-series (such as those used in finance), while the

design of `nitime` is more directly geared towards experimental data analysis. BrainVISA [Favre09], is a package that focuses on fMRI data and provides a complete array of analysis facilities, including preprocessing, visualization and batch workflows. In contrast, `nitime` is a developer-oriented library that focuses on implementation of algorithms and generic data-management objects. `Pyhrf` [Makni08] is a library which implements a joint detection-estimation approach to brain activity. Future development will hopefully lead to tighter integration of `nitime` with this library. Finally, `Nitime` time-series objects can use the newly introduced `datetime` data type in NumPy but do not depend on it, and can thus be used to manipulate any data set that fits into an n-dimensional NumPy array.

Importantly, analysis of fMRI data requires several steps of pre-processing, such as motion correction and file-format conversion to occur before this kind of analysis can proceed. Several software packages, such as BrainVISA, FSL [Smith04] , AFNI [Cox97] and SPM [Friston95] implement algorithms which can be used in order to perform these steps. Furthermore, the `nipype` library (`http://nipy.sourceforge.net/nipype/`), which is also part of the NiPy project, provides a Python programming interface to some of these packages. The design of `nitime` assumes that the data has been pre-processed and can be read in either as a NumPy `ndarray` or in the standard NIfTI file-format.

Next, we will describe the design of `nitime` and the decision-making process leading to this implementation. Then, we will demonstrate how `nitime` can be used to analyze real-world data.

Software design

Today, most high-level software uses object oriented (OO) design ideas to some extent. The Python language offers a full complement of syntactic support to implement OO constructs, from simple one-line classes to complex hierarchies. Previous experience has shown us that designing good OO interfaces is far, far harder than it appears at first. Most of us who have come to write software as part of our scientific work but without much formal training in the matter, often prove surprisingly poor performers at this task.

The simplest description of what an object is in computer science, presents it as the coupling of data and functions that operate on said data. The problem we have seen in the past with a literal interpretation of this description, however, is that it is very easy to build object hierarchies where the data and the algorithms are more tightly coupled than necessary, with numerical implementation details living inside the methods of the resulting objects, and the objects holding too much state that is freely reused by all methods. This effectively buries the algorithms inside of objects and makes it difficult to reuse them in a different design without carrying the containing objects.

To a good extent, this is the problem that the C++ Standard Template Library tries to address by separating containers from algorithms and establishing interfaces for generically coupling both at use time. For `nitime`, we have tried to follow this spirit by separating our implementation into three distinct parts:

1. A purely functional library, `nitime.algorithms`, that implements only the numerical part of time-series analysis algorithms. These functions manipulate NumPy arrays and standard Python types (integers, floats, etc.), which makes their calling signatures necessarily somewhat long, in the classical style of well known libraries such as LAPACK.

2. A set of "dumb" data container objects for time-series data, that do as little as possible. By forcing them to have a very minimal interface, we hope to reduce the chances of making significant design mistakes to a minimum, or of creating objects whose interface 'over-fits' to our needs and is thus not useful for others. In this respect, we try to follow the excellent example laid out by Python itself, whose core objects have small but remarkably general and useful interfaces.

3. A set of "smart" objects, which we have called *analyzers*, that provide access to the algorithms library via easy to use, high-level interfaces. These objects also assist in bookkeeping of state, and possibly caching information that may be needed in different computations (such as the Fourier Transform of a signal).

Our analyzer objects are a lightweight binding of #1 and #2 above, and thus represent a minimal investment of effort and code. If they prove to be a poor fit for a new user of `nitime` (or for us in the future), there is no significant loss of functionality and no major investment of time has been made for naught. The real value of the library lies in its algorithms and containers, and users should be free to adapt those to the task at hand in any way they see fit. We now provide a brief overview of these three components, whose use we will illustrate in the next section with a more detailed example.

Algorithms

The `nitime.algorithms` module currently implements the following algorithms:

Spectral transforms

Transforms of time-series data to the frequency-domain underlie many methods of time-series analysis. We expose previous implementations of spectral transforms taken from `mlab`, Matplotlib's library of numerical algorithms [Matplotlib]. In addition, we have written algorithms for the calculation of a standard periodogram and cross-spectral density estimates

based both on regular and multi-taper periodograms. The multi-taper periodogram was implemented here using discrete prolate spheroidal (Slepian) sequences ([NR07], [Percival93], [Slepian78]).

Coherency

Coherency is an analog of cross-correlation between two time-series calculated in the frequency domain, which can be used to study signals whose underlying spectral structure is similar despite containing substantial differences in the time domain. In fMRI analysis, this technique is used in order to calculate the functional connectivity between time-series derived from different voxels in the brain, or different ROIs and in order to infer the temporal relations them [Sun05]. One of the inherent problems in the analysis of functional connectivity of fMRI signals is that the temporal characteristics of the hemodynamic response in different areas of the brain may differ due to variations in the structure of the local vasculature in the different regions. Consequently, the delay between neural activity in a voxel and the peak of the ensuing influx of oxygenated blood may differ quite significantly between different voxels, even if the neural activity which is the root cause of the BOLD response and the quantity of interest, were identical. Thus, the correlation between the two time-series derived from the BOLD response in two different regions may be quite low, only because the hemodynamic response in one area begins much later than the hemodynamic response in the other area.

This limitation can be overcome to some degree, by conducting the analysis in the frequency domain, instead of the time domain. One type of analysis technique which examines the correspondence between two or more time-series in the frequency domain is coherency analysis. *Coherency* is defined as:

$$Coherency_{xy}(\nu) = \frac{f_{xy}(\nu)}{\sqrt{f_{xx}(\nu) f_{yy}(\nu)}}, \qquad (1)$$

where $f_{xy}(\nu)$ is the cross-spectral density between time-series x and time-series y in the frequency band centered on the frequency ν; $f_{xx}(\nu)$ and $f_{yy}(\nu)$ are the frequency-dependent power-spectral densities of time-series x and y respectively.

The squared magnitude of the coherency, known as *coherence*, is a measure of the strength of the functional coupling between x and y. It varies between 0 and 1 and will be high for two time-series which are functionally coupled even if the delays in their respective hemodynamic responses differ substantially. The phase $\phi(\nu)$ of the coherency can relay the temporal delay between the two time-series, via the relation $\Delta t(\nu) = \phi(\nu)/(2\pi\nu)$.

Importantly, the temporal resolution at which the delay can be faithfully reproduced in this method does not depend on the sampling rate (which is rather slow in fMRI), but rather depends on the reliability with which the hemodynamic response is produced given a particular activity. Though the hemodynamic response may vary between different subjects and between different areas in the same subject, it is rather reproducible for a given area in a given subject [Aguirre98].

In our implementation, these quantities can be computed with various methods to extract the cross-spectral density f_{xy} and the spectral densities f_{xx} and f_{yy}.

Regularized coherency

In addition to the standard algorithm for computing coherency, we have implemented a regularized version, which permits robust computation of coherence in the presence of small denominators (that occur for frequency bands where f_{xx} or f_{yy} is very small). Omitting the frequency ν for notational brevity, we replace eq. (1) with:

$$Coh^{R}_{xy,\alpha\epsilon} = \frac{|\alpha f_{xy} + \epsilon|^2}{\alpha^2 (f_{xx} + \epsilon)(f_{yy} + \epsilon)}, \qquad (2)$$

where α and ϵ are real numbers. This expression tends to Coh_{xy} when $\epsilon \to 0$, but is less sensitive to numerical error if either f_{xx} or f_{yy} is very small. Specifically, if $|f| \gg \epsilon$ then $Coh^{R}_{xy,\alpha\epsilon} \to Coh_{xy}$ (where f is any of f_{xx}, f_{yy} or f_{xy}), and if $f \approx \epsilon$ then:

$$Coh^{R}_{xy,\alpha\epsilon} \to \frac{(\alpha+1)^2}{4\alpha^2} Coh_{xy} \approx \frac{1}{4} Coh_{xy} \qquad (3)$$

for $\alpha \gg 1$. We note that this is only an order-of-magnitude estimate, not an exact limit, as it requires replacing ϵ by f_{xx}, f_{yy} and f_{xy} in different parts of eq. (1) to factor out the Coh_{xy} term.

For the case where $|f| \ll \epsilon \ll \alpha$, $Coh^{R}_{xy,\alpha\epsilon} \to \frac{1}{\alpha^2}$. In this regime, which is where small denominators can dominate the normal formula returning spurious large coherence values, this approach suppresses them with a smooth decay (quadratic in α).

Event-related analysis

A set of algorithms for the calculation of the correspondence between fMRI time-series and experimental events is available in `nitime`. These are univariate statistics calculated separately for the time-series in each voxel or each ROI. We have implemented a standard least squares estimate of the hemodynamic response function in response to a series of different events [Dale00].

In addition, we have implemented a fast algorithm for calculation of the cross-correlation between a series of events and a time-series and comparison of the resulting event-related response functions to the baseline variance of the time-series.

Containers

A `TimeSeries` object is a container for an arbitrary n-dimensional array of data (a NumPy `ndarray` object), along with a single one-dimensional array of time points. In the data array, the first $n-1$ dimensions are considered to describe the data elements (if $n = 1$, the elements are simply scalars) and the last dimension is the time axis. Since the native storage order of NumPy arrays is C-style (last index varies fastest), our choice gives greater data locality for algorithms that require taking elements of the data array and iterating over their time index. For example, a set of recordings from a multichannel sensor can be stored as a 2-d array A, with the first index selecting the channel and the second selecting the time point. In C-order storage, the data for channel i, $A[i]$ will be contiguous in memory and operations like an FFT on it will benefit from cache locality.

The signature of the `UniformTimeSeries` constructor is:

```
def __init__(self, data, t0=None,
             sampling_interval=None,
             sampling_rate=None,
             time=None, time_unit='s')
```

Any attribute not given at initialization time is computed at run time from the others (the constructor checks to ensure that sufficient information is provided, and raises an error otherwise). The standard Python approach for such problems is to use properties, but properties suffer from the problem that they involve a call to a getter function on every access, as well as requiring explicit cache management to be done in the getter. Instead, we take advantage of the dynamic nature of Python to find a balance of property-like delayed evaluation with attribute-like static storage.

We have defined a class called `OneTimeProperty` that exposes the descriptor protocol and acts like a property but, on first access, computes a value and then sets it statically as an instance attribute. The function is then called only once, and any further access to the name requires only a normal, static attribute lookup with no overhead. The code that implements this idea, stripped of comments and docstrings for the sake of brevity but otherwise complete, is:

```
class OneTimeProperty(object):
   def __init__(self,func):
       self.getter = func
       self.name = func.func_name

   def __get__(self,obj,type=None):
       if obj is None:
           return self.getter
       val = self.getter(obj)
       setattr(obj, self.name, val)
       return val
```

When writing a class such as `UniformTimeSeries`, one then declares any property whose first computation should be done via a function call using this class as a decorator. As long as no circular dependencies are introduced in the call chain, multiple such properties can depend on one another. This provides for an implicit and transparent caching mechanism. Only those attributes accessed, either by user code or by the computation of other attributes, will be computed. We illustrate this with the implementation of the `time`, `t0`, `sampling_interval` and `sampling_rate` attributes of the `UniformTimeSeries` class:

```
@OneTimeProperty
def time(self):
    npts = self.data.shape[-1]
    t0 = self.t0
    t1 = t0+(npts-1)*self.sampling_interval
    return np.linspace(t0,t1,npts)

@OneTimeProperty
def t0(self):
    return self.time[0]

@OneTimeProperty
def sampling_interval(self):
    return self.time[1]-self.time[0]

@OneTimeProperty
def sampling_rate(self):
    return 1.0/self.sampling_interval
```

We have found that this approach leads to very readable code, that lets us delay computation where desired without introducing layers of management code (caching, private variables for getters, etc.) that obscure the main intent.

We have so far overlooked one important point in our discussion of "automatic attributes": the case where the quantities depend on mutable data, so that their previously computed values become invalid. This is a problem that all caching mechanisms need to address, and in its full generality it requires complex machinery for cache state control. Since we rely on an implicit caching mechanism and our properties become regular attributes once computed, we can not use regular cache dirtying procedures. Instead, we have provided a `ResetMixin` class that can be used for objects whose automatic attributes may become invalid. This class provides only one method, `reset()`, that resets *all* attributes that have been computed back to their initial, unevaluated state. The next time any of them is requested, its accessor function will fire again.

Analyzers

We now describe our approach to exposing a high-level interface to the analysis functions. We have constructed a set of lightweight objects called *analyzers*, that group together a set of conceptually related analysis algorithms and apply them to a specific time-series object. These objects have a compact implementation and no significant numerical code; their purpose is to do simple book-keeping and to allow for end-user code that is readable and compact. Their very simplicity also means that they shouldn't be judged too severely if they don't fit a particular application's needs: it is easy enough to implement new analysis objects as needed. We do hope that the ones provided by `nitime` will serve many common cases, and will also be useful

reference implementations for cases that require writing new ones.

All analyzers follow a similar pattern: they are instantiated with a `TimeSeries` object of interest, which they keep an internal reference to, and they expose a series of attributes and methods that compute specific algorithms from the library for this time series. For all the main quantities of interest that have a static meaning, the analyzer exposes an attribute accessor that, via the `OneTimeProperty` class, calls the underlying algorithm with all required parameters and stores the result in the attribute for further use. In addition to these automatic attributes, analyzers may also expose normal methods that provide simplified interfaces (with less parameters) to algorithms. If any of these methods requires one of the automatic attributes, it will be naturally computed by the accessor on first access and this result will be then stored in the instance. We will now present examples showing how to analyze both synthetic and real fMRI data with these objects.

Examples: coherency analysis

Analysis of synthetic time-series

The first example we present is a simple analysis stream on a pair of synthetic time-series (Figure 2), of the form

$$x(t) = \sin(\alpha t) + \sin(\beta t) + \epsilon_x \quad (4)$$

$$y(t) = \sin(\alpha t + \phi_1) + \sin(\beta t - \phi_2) + \epsilon_y \quad (5)$$

where $\epsilon_{x,y}$ are random Gaussian noise terms and $\phi_i > 0$ for $i = 1, 2$, such that each is a superposition of two sinusoidal functions with two different frequencies and some additional uncorrelated Gaussian white noise and the relative phases between the time-series have been set such that in one frequency, one series leads the other and in the other frequency the relationship is reversed.

We sample these time series into an array `data` from which a `UniformTimeSeries` object is initialized:

```
In [3]: TS = UniformTimeSeries(data,sampling_rate=1)
```

A correlation analyzer object is initialized, using the time-series object as input:

```
In [4]: Corr = CorrelationAnalyzer(TS)
```

`Corr.correlation` now contains the full correlation matrix, we extract the position `[0,1]`, which is the correlation coefficient between the first and the second series in the object:

```
In [5]: Corr.correlation[0,1]
Out[5]: 0.28727768
```

The correlation is rather low, but there is a strong coherence between the time-series (Figure 2B) and in particular in the two common frequencies. We see this by initializing a coherence analyzer with the time-series object as input:

```
In [6]: Coh = CoherenceAnalyzer(TS)
```

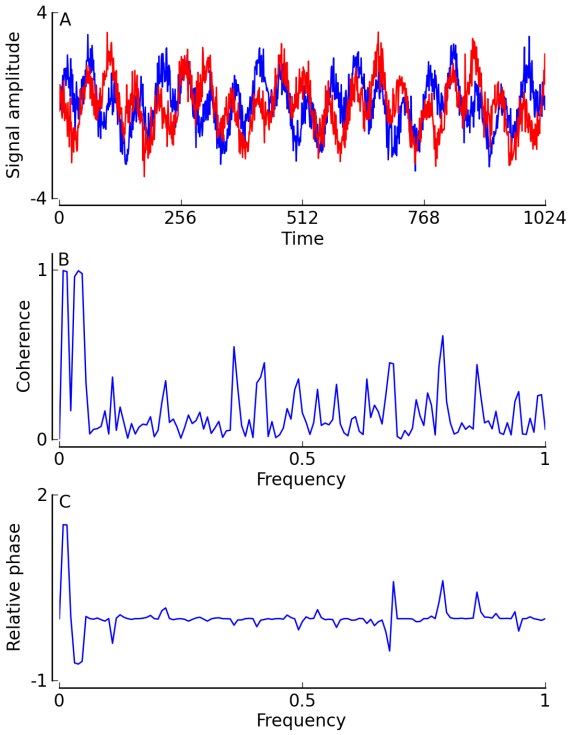


Figure 2. Coherency analysis - an example: ***A****: two synthetic time-series are displayed.* ***B****: The coherence is displayed as a function of frequency.* ***C****: The coherency phase-delay between the time-series is presented, as a function of frequency.*

We examine specifically the coherence in the frequency bands of interest with indices 2 and 6:

```
In [7]: Coh.coherence[0,1,2]
Out[7]: 0.9893900459215027

In [8]: Coh.coherence[0,1,6]
Out[8]: 0.97800470864819844
```

These two high coherence values are what gives rise to the two prominent peaks in the coherence in Figure 2B. In addition, the relative phases are reversed in the two frequencies composing these time series. This is reflected in the relative phase between the time-series (Figure 2C), which can be calculated in a similar way.

Analysis of fMRI data

Our second example (Figure 3) demonstrates the analysis of actual experimental fMRI data, acquired by David Bressler and Michael Silver. In this experiment, subjects fixated their gaze on a dot at the center of the visual field and viewed a wedge-shaped section of a circle (like a pizza slice), which slowly rotated around the fixation dot at a rate of one full cycle every 32 seconds. The wedge contained a flickering checker-board

pattern. This pattern of stimulation is known to stimulate activity in visual areas, which can be measured with fMRI. In half of the scans, the subject was instructed to detect the appearance of targets inside the checker-board pattern. In the other half of the scans, the subject was instructed to detect targets appearing in the fixation point at the center of gaze. Thus, in both cases, the subject's attention was engaged in a difficult detection task (tasks were adjusted so that the difficulty in both cases was similar). The only difference between the two conditions was whether attention was directed into the area covered by the checker-board wedge or out of this area. Previous research [Lauritzen09] has shown that allocation of attention tends to increase coherence between areas in early visual cortex and between areas in visual cortex and IPS areas (see Figure 1). Data was recorded from subjects' brains in the scanner, while they were performing this task. Visual ROIs were defined for each subject. These ROIs contain the parts of cortex which represent the areas of the visual field through which the checker-board wedge passes in its rotation, but not the area of the visual field close to the center of the focus of the gaze, the area in which the fixation point is presented. Thus, the activity measured in the experiment is in response to the same visual stimulus of the checker-board wedge; in half the scans, while attention is directed to the wedge and in the other half, when attention is directed away from the wedge.

In order to examine the functional connectivity between the ROIs, we start from data stored on disk in a `.npy` file containing an array with time-series objects, created from the raw fMRI data for a single subject:

```
In [1]: tseries_arr = np.load('tseries.npy')
```

Each `TimeSeries` object in this array corresponds to the data for a separate scan, and it contains the mean BOLD data for 7 separate ROIs (one per visual area of interest, see Figure 3). Attention was directed to the fixation point in the even scans and to the checker-board wedge in the odd scans. We initialize coherence analyzers for each of the scans and store those in which attention was directed to the wedge separately from those in which attention was directed to the fixation point:

```
In [2]: C_fix = map(CoherenceAnalyzer,
   ....:            tseries_arr[0::2])  # even scans

In [3]: C_wedge = map(CoherenceAnalyzer,
   ....:              tseries_arr[1::2])  # odd scans
```

We extract the cross-coherence matrix for all the ROIs in one frequency band (indexed by 1) and average over the scans:

```
In [4]: mean_coh_wedge = array([C.coherence[:,:,1]
   ....:   for C in C_wedge]).mean(0)

In [5]: mean_coh_fix = array([C.coherence[:,:,1]
   ....:   for C in C_fix]).mean(0)
```

In order to characterize the increase in coherence with attention to the wedge, we take the difference of the resulting array:

```
In [6]: diff = mean_coh_wedge - mean_coh_fix
```

In Figure 3, we have constructed a graph (using NetworkX [NetworkX]) in which the nodes are the visual area ROIs, presented in Figure 1. The edges between the nodes represent the *increase* in coherence in this frequency band, when subjects are attending to the wedge, relative to when they are attending to the appearance of targets in the fixation point. This graph replicates previous findings [Lauritzen09]: an increase in functional connectivity between visual areas, with the allocation of voluntary visual attention to the stimulus.

These examples demonstrate the relative simplicity and brevity with which interactive data analysis can be conducted using the interface provided by `nitime`, resulting in potentially meaningful conclusions about the nature of the process which produced the time-series. This simplicity should facilitate the study of complex data sets and should enable more sophisticated methods to be developed and implemented.

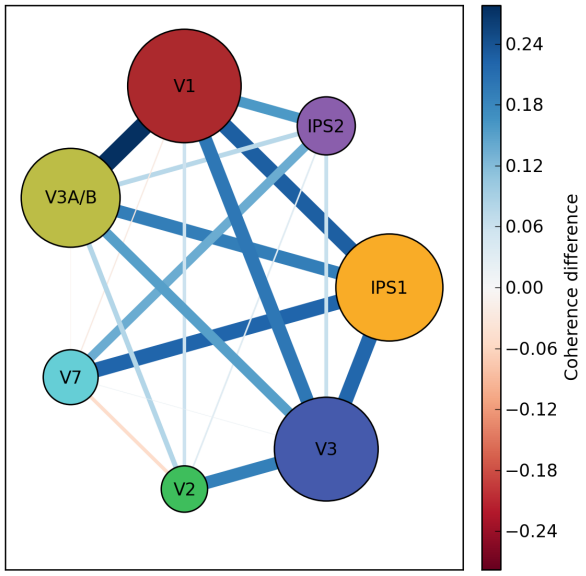


Figure 3. Functional connectivity in the visual cortex: *In the graph presented, the nodes represent the areas of the brain described in Figure 1 (node colors and labels match those of Figure 1).*

Summary and outlook

We have introduced `nitime`, a library for the analysis of time-series data from neuroimaging experiments and in particular from fMRI experiments, developed as part of the NiPy project. `Nitime` provides implementations of several algorithms, including coherency analysis, and a high-level interface for interaction with time-series data. Its design emphasizes a decoupling of algorithmic implementation and object-oriented features. This is meant to facilitate use of the algorithms in contexts other than neuroscience and contributions

from developers in other fields. Future developments will include implementations of additional algorithms for calculation of bivariate and univariate quantities, as well as tools for visualization of time-series.

Acknowledgments

We would like to thank the NiPy development team, in particular Matthew Brett for many helpful discussions, Gaël Varoquaux, for discussions on properties that led to `OneTimeProperty` and Dav Clark for feedback on the time-series object interface. We are thankful to David Bressler and Michael Silver for providing us with experimental data, and to all the members of Michael Silver's lab for comments and feedback on this work. Felice Sun, Thomas Lauritzen, Emi Nomura, Caterina Gratton, Andrew Kayser, Ayelet Landau and Lavi Secundo contributed Matlab code that served as a reference for our implementation of several algorithms. Finally, we'd like to thank Mark D'Esposito, director of the NiPy project, for supporting the development of nitime as part of the overall effort. NiPy is funded by the NIH under grant #5R01MH081909-02.

References

[Aguirre97] Aguirre GK, Zarahn E, D'Esposito M (1997). *Empirical Analyses of BOLD fMRI Statistics II. Spatially Smoothed Data Collected under Null-Hypothesis and Experimental Conditions.* Neuroimage **5**: 199-212.

[Aguirre98] Aguirre GK, Zarahn E, D'esposito M (1998). *The variability of human, BOLD hemodynamic responses.* Neuroimage **8**: 360-9.

[BSD] *The BSD Software License.* http://www.opensource.org/licenses/bsd-license.php.

[Cox97] Cox RW and Hyde JS (1997). *Software tools for analysis and visualization of FMRI data.* NMR in Biomedicine, **10**: 171-178.

[Dale00] Dale, AM (2000). *Optimal Experimental Design for Event-Related fMRI.* Human Brain Mapping **8**: 109-114.

[Favre09] Favre L, Fouque A-L, et al. (2009) *A Comprehensive fMRI Processing Toolbox for BrainVISA.* Human Brain Mapping **47**: S55.

[Friston94] Friston, KJ (1994). *Functional and Effective Connectivity in Neuroimaging: A Synthesis.* Human Brain Mapping **2**: 56-78.

[Friston95] Friston KJ, Ashburner J, et al. (1995) *Spatial registration and normalization of images.* Human Brain Mapping, **2**: 165-189.

[Huettel04] Huettel, SA, Song, AW, McCarthy, G (2004). *Functional Magnetic Resonance Imaging.* Sinauer (Sunderland, MA).

[Kayser09] Kayser AS, Sun FT, D'Esposito M (2009). *A comparison of Granger causality and coherency in fMRI-based analysis of the motor system.* Human Brain Mapping, *in press.*

[Lauritzen09] Lauritzen TZ , D'Esposito M, et al. (2009) *Top-down Flow of Visual Spatial Attention Signals from Parietal to Occipital Cortex.* Journal of Vision, *in press.*

[Makni08] Makni S, Idier J, et al. (2008). *A fully Bayesian approach to the parcel-based detection-estimation of brain activity in fMRI.* Neuroimage **41**: 941-969.

[Matplotlib] Hunter, JD (2007). *Matplotlib: A 2D graphics environment.* Comp. Sci. Eng. **9**: 90-95.

[NetworkX] Hagberg AA, Schult DA, Swart PJ (2008). *Exploring network structure, dynamics, and function using NetworkX.* in Proc. 7th SciPy Conf., Varoquaux G, Vaught T, and Millman J (Eds), pp. 11–15.

[NiPy] Millman KJ, Brett M (2007). *Analysis of functional Magnetic Resonance Imaging in Python.* Comp. Sci. Eng. **9**: 52-55.

[NR07] Press, WH, Teukolsky, SA, Vetterling, WT, Flannery, BP (2007). *Numerical Recipes: The Art of Scientific Computing.* 3rd edition, Cambridge University Press (Cambridge, UK).

[Percival93] Percival DB and Walden, AT (1993). *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques.* Cambridge: Cambridge University Press.

[Poldrack06] Poldrack, R (2006). *Region of interest analysis for fMRI.* Soc. Cog. Aff. Neurosci., **2**: 67-70.

[Silver05] Silver MA, Ress D ,Heeger DJ (2005). *Topographic maps of visual spatial attention in human parietal cortex.* J Neurophysiol, **94**: 1358-71.

[Slepian78] Slepian, D (1978). *Prolate spheroidal wave functions, Fourier analysis, and uncertainty V: The discrete case.* Bell System Technical Journal, **57**: 1371-1430.

[Smith04] Smith SM, Jenkinson M, et al. (2004). *Advances in functional and structural MR image analysis and implementation as FSL.* NeuroImage, **23**: 208-219.

[Sun05] Sun FT, Miller LM, D'Esposito M (2005). *Measuring interregional functional connectivity using coherence and partial coherence analyses of fMRI data.* Neuroimage **21**: 647-658.

[TimeSeries] Gerard-Marchant P, Knox M (2008). *Scikits.TimeSeries: Python time series analysis.* http://pytseries.sourceforge.net.

[Wandell07] Wandell BA, Dumoulin, SO, Brewer, AA (2007). *Visual field maps in human cortex.* Neuron **56**: 366-83.

Multiprocess System for Virtual Instruments in Python

Brian D'Urso (dursobr@pitt.edu) – *University of Pittsburgh, Department of Physics and Astronomy, 3941 O'Hara St., Pittsburgh, PA 15260* US

Programs written for controlling laboratory equipment and interfacing numerical calculations share the need for a simple graphical user interface (GUI) frontend and a multithreaded or multiprocess structure to allow control and data display to remain usable while other actions are performed. We introduce Pythics, a system for running "virtual instruments", which are simple programs typically used for data acquisition and analysis. Pythics provides a simple means of creating a virtual instrument and customizing its appearance and functionality without the need for toolkit specific knowledge. It utilizes a robust, multiprocess structure which separates the GUI and the back end of each instrument to allow for effective usage of system resources without sacrificing functionality.

Python is an attractive language for scientific programming because of the simplicity of expressing mathematical ideas and algorithms within it. However, it is the broad range of open source libraries available that enables scientific computing within Python. With the capabilities of libraries such as Numpy [numpy] and SciPy [scipy] for numerical processing, SymPy [sympy] for symbolic manipulation, and Sage [sage] integrating them together, Python is in a strong position to handle a wide range of scientific computational problems.

However, in experimental sciences, where computer-based data acquisition and control is dominated by a small number of proprietary programming systems, Python is not such an obvious choice. These proprietary systems often make the task of simple experiment control easy, however they often don't scale well to complex experiments because they lack a well-designed, general purpose programming language. We present a system starting with a complete programming language, Python, rather than trying to develop another special purpose language. There are existing, mature Python libraries for supporting data acquisition and experiment control; perhaps most critically PyVISA [pyvisa], which provides a robust bridge to commercially-supported VISA libraries and the experiment hardware they can control. In practice we also make use of the Python Imaging Library (PIL) [pil] for image manipulation, wxPython [wxpython] as a user interface toolkit, and matplotlib [matplotlib] for plotting.

In addition to being able to communicate with instruments, modern software for data acquisition and control must be able to present a graphical user interface (GUI) for display of data as well as providing a means for the experimenter to interact with an experiment in progress. GUI toolkits such as wxPython provide a means to create a GUI, but are not tailored to the requirements of data acquisition, where communication with instruments often must be proceeding in parallel with GUI operation. Furthermore, students working on experiments may have little or no programming experience, so programming a multithreaded or multiprocess application may be beyond what they can or want to pursue.

Here we introduce Pythics (PYTHon Instrument Control System), a multiprocess system designed to make it straightforward to write software for data acquisition and control with Python. There are several important features which expedience has taught us are needed to produce a successful system. First, the system must be cross platform, at least supporting Linux and Microsoft Windows XP. While many developers prefer Linux, Windows XP is presently often the simplest choice for interfacing instruments because of the support provided by the instrument manufacturers. Second, we want to avoid excessive dependencies that could make it difficult to install the system and lead to bloated code; instead we prefer to make modular, optional, orthogonal features that can be used if the supporting libraries are available. The system must provide a simple means of specifying a GUI which does not require knowledge of the underlying GUI toolkit and must be easy to understand, even for an inexperienced programmer. Critically, it must be possible for the GUI to continue to function even when data acquisition or any communication with instruments is in progress. In most cases, this requires a multithreaded or multiprocess system. Yet, we do not want to require the user-programmer to have a thorough understanding of multithreaded programming. So, the system must handle the multithreaded or multiprocess structure transparently.

Similar to some commercial software, we call the code that runs within Pythics a virtual instrument (VI), and in general multiple VIs may be running within a single Pythics application. We require that there be a mechanism for sharing objects between VIs, for example for sharing data between VIs or synchronizing some operation. An additional requirement is that the system must be robust. It should be possible to terminate one VI without affecting the others, or at least to continue without having to restart Pythics.

Features

We are willing to accept some loss in GUI design flexibility for the simplicity of programming Pythics that we require. We looked for a means of specifying a GUI that would be simple, would layout in a usable manner across a wide variety of screen and window sizes, and

would grow to handle a VI GUI which might gradually increase in complexity over time as a VI evolves. We found inspiration for a solution in the layout of hypertext markup language (HTML) used in web browsers, which has proven to be remarkably flexible over time. While HTML is primarily oriented towards the layout of text, we primarily require the layout of GUI elements. Fortunately, we found adequate flexibility in extensible hypertext markup language (XHTML), a markup language similar to HTML which also conforms to extensible markup language (XML) syntax requirements. By following the stricter XHTML syntax, the implementation of the Pythics layout engine can be less forgiving and thus simpler than a modern web browser. Furthermore, we only support a minimal subset of XHTML as needed for layout of simple lines or tables of GUI elements, and a small number of cascading style sheets (CSS) attributes to customize the appearance of the GUI. In practice, we make extensive use of the `object` XHTML element to insert GUI elements.

Using an XHTML-based system for GUI layout already constrains many features of our layout engine. Elements (e.g. a button) are generally fixed in height and may or may not expand in width as the enclosing window is resized. As more elements are added, the GUI will generally have to be scrolled vertically for access to all the GUI elements, while horizontal scrolling is avoided unless necessary to fit in the minimum horizontal size of the GUI elements. It may seem surprising that a layout system based on a system designed for text (HTML) would be general enough to specify a GUI, but the adaptability of the world wide web to new functionality demonstrates the flexibility of HTML.

In a large and evolving experiment, it is typical to have an ever-growing number of VIs for data acquisition and control. To organize multiple VIs running within Pythics, we again borrow a feature from web browsers: tabbed browsing. In Pythics, each VI runs within its own tab, all contained within the same Pythics window. This help avoid confusion if many windows are open, and in particular if multiple instances of Pythics are running, a scenario which is both possible and, in some cases, desirable.

Pythics maintains a strict separation between the GUI layout specification and the VI functionality by separating the code for a VI into two or more files. The first file contains the XHTML specification of the GUI and additional objects which trigger loading of the remaining files. These files are pure python, loading any additional libraries which may be needed and defining the functions triggered by GUI callbacks.

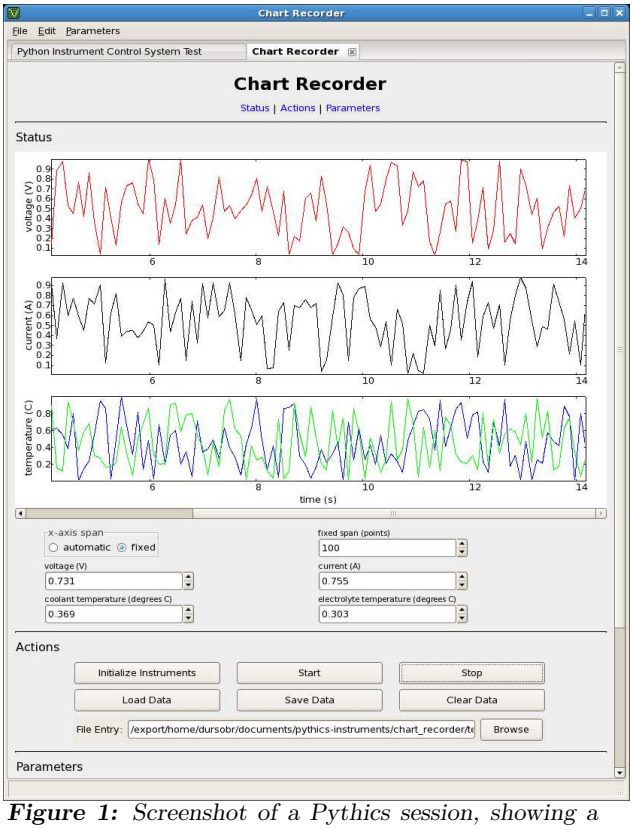


Figure 1: *Screenshot of a Pythics session, showing a chart recorder VI.*

Multiprocess Structure

The multiprocess structure of Pythics is dictated by the GUI structure and the requirements of the VIs. First, many GUI toolkits (including wxPython) place the GUI for all VIs within a single Pythics application to reside in the same thread of the same process. Since each VI may have its own tasks to work on while the GUI is to remain functional, each VI must also have its own worker thread or process. An early version of Pythics used a separate thread for each VI, but we found that the system was excessively fragile, with a fault in a single VI sometimes ending in a abandoned, but often still running thread, because of the limited methods to safely kill a thread in Python. The need for a more robust system and the appearance of the multiprocessing module in Python 2.6 lead us to a multiprocess design, where each VI has its own worker process which handles data acquisition and data processing. Furthermore, the use of multiple processes avoids the Python global interpreter lock (GIL), which could limit the benefits of using multiple threads alone.

For simplicity, we provide only a single thread in a single process for the work of each VI, although each VI is in principle free to use and manage multiple processes or threads as needed. Since wxPython restricts the GUI to a single process and thread, we are pushed towards one particular process structure: each VI has its own worker process which communicates with a single shared GUI process. If multiple GUI processes are desired, multiple instances of Pythics can be running simultaneously, although there is no support for communication between VIs in different Pythics instances. For the rest of the description here, we assume there is only a single running Pythics instance.

The multiprocess structure of Pythics is illustrated in

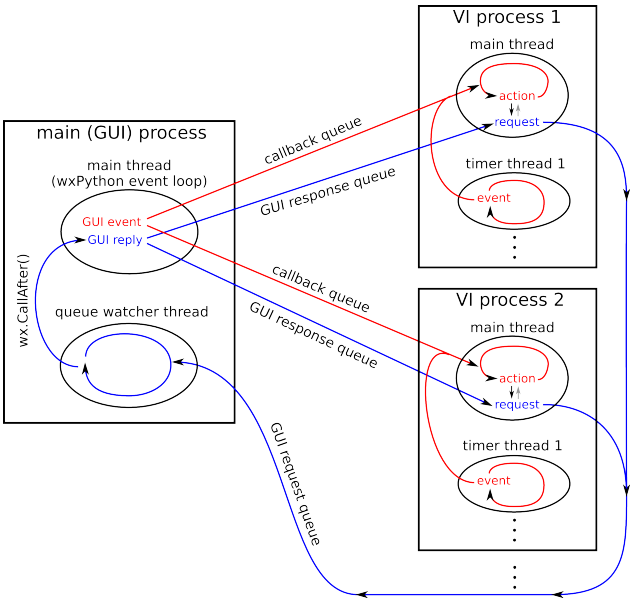


Figure 2: *The multiprocess structure of Pythics.*

Figure 2. Each VI worker process communicates back and forth with the single GUI process by placing messages in queues. There are three routes for communication between the worker processes and the GUI process. First, the GUI may want to trigger a callback function in response to an action, for example a button being pressed. Each worker process has a queue for callback requests, and each worker process waits for a request on this queue when it is idle. Next, a worker process, typically within a callback function, may want the GUI to respond in some way or may want to get some other information from the GUI, for example reading the value of a parameter from within a box in the GUI panel. The GUI process has a thread which watches a single queue where requests from all worker processes are funneled. This thread waits for a request to be appear in the queue, and when one arrives, the thread transfers the request to the wxPython main thread with a call to `wx.CallAfter`. The GUI process then answers the request by placing the response in the response queue for the requesting worker process. Finally, the worker process receives the response and continues with its task.

In practice, the Pythics user-programmer does not have to worry about the inter-process message passing and queue structure. The execution of callback functions is handled automatically. Requests to and responses from the GUI process are handled by GUI control proxy objects, essentially encapsulating all of the multiprocess and queue structure. We will go into further details in later sections.

GUI Specification

The GUI layout engine, which allows the arrangement of GUI elements within a VI GUI tab, was inspired by, and originally based on, the wxPython HtmlWindow. This wxPython widget can layout HTML text mixed with GUI elements such as buttons. We eventually wrote a replacement XHTML renderer for Pythics so it could gracefully handle exceptions in the process of creating controls. If a Python exception is encountered while creating a GUI element in our renderer, the element is replaced by a red box which displays the exception, making debugging much easier.

The GUI specification or markup language is XML, and is essentially a subset of XHTML. The style of the text elements, the background color, etc. are specified with a simplified cascading style sheets (CSS) system, also similar to typical XHTML or HTML files. Our specification must have many GUI elements that are not common for XHTML, so we make extensive use of the XHTML `object` element. Within the object element, we use the `classid` attribute to specify the GUI element class (e.g. `Button`), and the `id` attribute to specify a name for the GUI element which becomes the name of the object in Python. Other parameters for the GUI elements are specified with `param` elements within the object element.

An example GUI, here for a “Hello World” VI in Pythics, illustrates how a GUI is specified:

```
<html>
    <head><title>Hello World</title></head>
    <body>

        <h1>Hello World</h1>

        <object classid='Button' width='200'>
            <param name='label' value='Run'/>
            <param name='action' value='hello_world.run'/>
        </object>
        <br/>

        <object classid='TextBox' id='result' width='200'>
        </object>
        <br/>

        <object classid='ScriptLoader' width='100%'>
            <param name='filename' value='hello_world'/>
        </object>

    </body>
</html>
```

In the `<head>` element, the `<title>` element gives a title to the VI which appears on the tab containing the VI in the Pythics GUI. There is then a printed title within the `<h1>` element, followed by the primary GUI elements, a button and a text box. Note the `param` element with `name='action'` within the button object. This specifies the callback function which is executed when the button is pressed. We have also given the text box a name, through its `id` attribute. We will use this to access the text box for displaying a message later.

Without the final `object` element, Pythics would not know where to find the callback function specified by the button object. The `ScriptLoader` object functions similar to the Python `import` statement, loading a file which typically contains callback functions. This object can be used multiple times to import multiple files. This object also shows up in the GUI as a line of

text to show that it is there. The resulting GUI, which appears within a tab in Pythics, is shown in Figure 3.

Figure 3: *Screenshot of example GUI window for the "Hello World" example.*

Other GUI elements that are available within Pythics include images, many kinds of buttons, file dialogs, numeric input/output boxes, spreadsheets, sliders, embedded Python shells, and plots (using wxPython or matplotlib).

Callback Functions

Our "Hello World" example doesn't yet have any functionality. To make it respond to pressing the button, we need to introduce callback functions within Pythics. We want the structure of callback functions in Pythics to be as unconstrained as possible, but they do need some way to access the GUI elements. We do not introduce a formal event object, since this adds complexity that is unnecessary in most VIs. If a callback function needs information about the event that triggered it, it should get that information by addressing the appropriate GUI element.

In order give callback functions access to the GUI elements, we introduce a simple calling convention. The callback function is called with all of the GUI elements which have an `id` attribute in the XML specification file, which we will call named elements, as keyword arguments. There are no other arguments passed to callback functions. Thus, a callback function could receive all the named elements as a dictionary using the Python `**kwargs` syntax, or it can separate out the named elements it will use as individual arguments and group the unused named elements together with the `**kwargs` syntax.

The completion of our "Hello World" example clarifies the use of callback functions. Here is the entire Python file for our example:

```
def run(result, **kwargs):
    result.value = "Hello, world!"
```

Note that the callback function receives the one named element, `result`, as a keyword argument, and any other named elements (there are none in this case) would be grouped into `kwargs` as a dictionary. Thus, more GUI elements can be added without breaking this callback function. We display the message "Hello, world!" within the text box in the GUI with a simple call to to the GUI element proxy, by setting its `value` attribute. Clearly, no knowledge of the multiprocess structure and message passing within Pythics is needed. In most cases, these proxy objects have a fairly high level interface within Pythics, so most exchanges with the GUI within the callback functions require only a small number of commands using the GUI element proxies.

Other than our callback function calling convention, the files that contain the callback functions are standard, pure Python files. Additional functions can be defined, and packages can be imported as needed. Pythics itself does not require any imports within the callback function files, making it easy to maintain a clean namespace.

Additional Features

There are several other features of Pythics which make it more useful for making functional VIs. Most of these are implemented as additional or modified GUI controls, so they fit easily into the framework described above. These include:

- The values entered into a VI's GUI controls can be stored in a parameters file. This includes support for default parameters, which are automatically loaded when a VI is started. It also allows for alternative sets of parameters which can be saved and recalled.
- Global namespaces are available for sharing data between VIs. This uses the Python multiprocessing `Namespace` object, so almost arbitrary objects can be shared.
- Optional display of images in the GUI with shared memory instead of message passing. With one option in the GUI specification, the image display proxy will pass data to its control using shared memory to allow for much more rapid updates. In the current implementation, the maximum size of the image must be given in the GUI specification if shared memory is used.
- Timer elements are available for executing a function at regular intervals in time. To the user-programmer, a timer looks like any other GUI element, such as a button, which calls its action callback at regular intervals. Timers actually operate completely in the worker process through their proxy object, but within a separate thread from the callback function execution. This structure eliminates the overhead of message passing for frequently executed callbacks when no GUI interaction is needed. Timers are implemented with Python threading Event objects, so they can be interrupted at any time. A single VI may have many timers.

- There is a `SubWindow` element which can embed an entire second VI within a box inside the GUI of the first VI, allowing for modularization of GUIs as well as functions. There are methods to pass data between the fist and second VI, although both run in the same thread and process.

Future

Pythics is still evolving very quickly, and we anticipate continuing to add features, primarily driven by the needs of our laboratory or the needs of other groups that may start to use Pythics. The highest priority item is documentation, which will be necessary for the Pythics community to grow. As a more technically challenging direction, we may pursue operating Pythics over a network, with the GUI and worker processes on different machines communicating over the network. There is already support for this in the Python multiprocessing module.

We plan continued expansion and improvement in the graphics display controls available within Pythics. One significant required feature is the ability to plot data in near real time, as it is being acquired. Simple, fast, line plots are available now in Pythics, but more complex plots require matplotlib, which is often too slow for keeping up with incoming data when most of the available processor time is used acquiring and processing data. We are investigating alternative plotting libraries which could be easily used within wxPython. Another graphics feature we would like to add to Pythics is simple three-dimensional display capability, similar to VPython [vpython]. This could make Pythics an attractive system to use for teaching introductory computational physics classes, both for data acquisition and for simple models and calculations.

We also plan to improve the GUI layout design tools available for Pythics, to make it easier to write the XML GUI specification. One possibility would be to have a graphical editor for the GUI interfaces, and perhaps a graphical HTML or XML editor could be adapted to this purpose. An alternative possibility, which is inspired by the reStructuredText format [rest] commonly used for Python documentation, would be to have an "ASCII art" layout description. In this case, we would develop a more readable text format for specifying and placing GUI elements, then translate that format to the XML format actually used by Pythics. In principle, there is no reason not to support both of these options.

We currently release Pythics under the GNU General Public License (GPL), and it is free to download [pythics]. We hope to attract the interest of other researchers and students, and that they will contribute to the success of Pythics. Ultimately, Pythics could become a center for free exchange of code, instrument drivers, and data analysis tools.

References

[numpy] T. Oliphant et al., *NumPy*, http://numpy.scipy.org/

[scipy] E. Jones, T. Oliphant, P. Peterson, et al. SciPy http://www.scipy.org/

[sympy] Development Team (2008). SymPy: Python library for symbolic mathematics http://code.google.com/p/sympy/

[sage] W. Stein et al., *Sage Mathematics Software*, http://www.sagemath.org/

[pyvisa] http://pyvisa.sourceforge.net/

[pil] http://www.pythonware.com/products/pil/

[wxpython] http://www.wxpython.org/

[matplotlib] J.D. Hunter. *Matplotlib: A 2D graphics environment.* Computing in Science and Engineering. (2007) 9: 90-95. http://matplotlib.sourceforge.net

[vpython] http://vpython.org/

[rest] http://docutils.sourceforge.net/rst.html

[pythics] http://code.google.com/p/pythics/

Neutron-scattering data acquisition and experiment automation with Python

Piotr A. Zolnierczuk (zolnierczukp@ornl.gov) – *Oak Ridge National Lab,* USA
Richard E. Riedel (riedelra@ornl.gov) – *Oak Ridge National Lab,* USA

PyDas is a set of Python modules that are used to integrate various components of the Data Acquisition System at Spallation Neutron Source (SNS). PyDas enables customized automation of neutron scattering experiments in a rapid and flexible manner. It provides wxPython-based GUIs for routine experiments as well as IPython command line scripting environment. Matplotlib and NumPy are used for data presentation and simple analysis. PyDas is currently used on a number of SNS instruments and plans exist to deploy it on new ones as they come on-line.

Introduction

The neutron is a useful probe of matter as the wavelengths of the so called cold neutrons[1] are on the order of inter-atomic distances in solids and liquids [SQU]. It has no charge and therefore it can interact with the atomic nuclei, for example with the proton in hydrogen that is virtually transparent to the X-ray radiation. Moreover, the energies of cold neutrons are on the same order as many excitations in condensed matter. And finally the neutron possesses a magnetic moment which means that it can interact with the unpaired electrons in magnetic atoms.
Using neutrons scientists can glean details about the nature of materials ranging from liquid crystals to superconducting ceramics, from proteins to plastics, and from metals to micelles and metallic glass magnets [PYN].

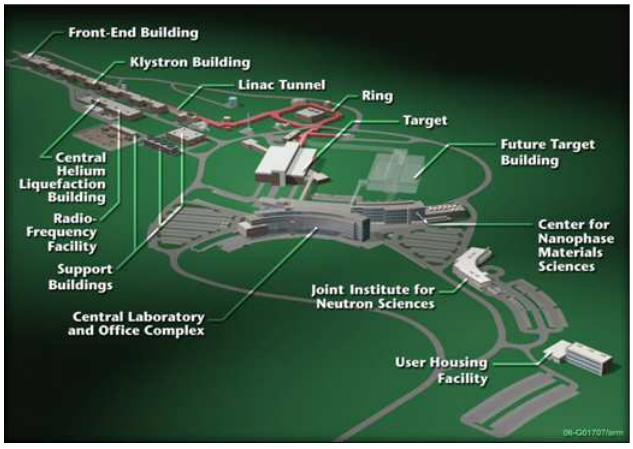


Figure 1. *Layout of the Spallation Neutron Source*

[1] Cold neutrons energies range from about 0.05 to about 25 meV, and their corresponding wavelengths range from 0.18 to 4.0 nm.

Basics of neutron scattering

Cold neutrons have to be produced either in a nuclear reactor or with the help of particle accelerators. The Spallation Neutron Source (SNS) at Oak Ridge National Laboratory, Tennessee is an accelerator-based neutron source that currently holds the Guinness World Record as the world most powerful pulsed spallation neutron source [ORN]. The neutrons are produced when a beam of very energetic protons (1GeV) bombards a liquid mercury target and in the process some neutrons are "spalled" or knocked out of mercury nucleus. Other neutrons are evaporated from the bombarded nucleus as it heats up. For every proton striking the nucleus about 20 to 30 neutrons are expelled. The neutrons emerging from the mercury target are too fast to be useful to study properties of materials. They need to be slowed down (or cooled) by passing through the moderator material - for example liquid hydrogen kept at the temperature of 20 K. The cold neutrons are then guided by a set of beam lines to specialized instruments.
The beam of neutrons then undergoes collimation, focusing and velocity selection (chopping) and is aimed at a sample. While many neutrons will pass through the sample, some will interact and bounce away at an angle. This scattering of neutrons yields important information about the positions, motions and magnetic properties of atoms in materials. Figure 2 shows a schematic lay-out of a neutron scattering experiment.

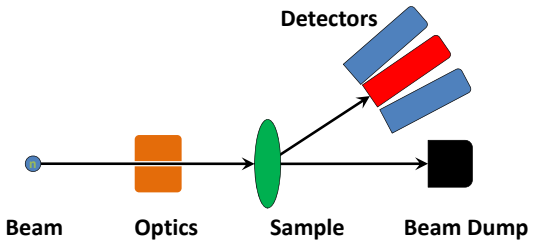


Figure 2. *Basic Elements of a neutron scattering experiment*

A number of devices and controls is required for a successful neutron scattering experiment. Such a set-up, conventionally called a neutron scattering instrument, needs first and foremost a neutron detector to record

the scattered neutrons. In most cases gaseous linear position-sensitive tubes or scintillator detectors are used [KNO]. In addition each instrument requires sample environment controls (temperature, pressure, and magnetic field), neutron beam optics controls (slits for collimation and choppers for velocity selection) and motors for mechanical alignment. An example of a real life neutron scattering instrument [CNC] is shown Figure 3.

A typical neutron experimental technique is called scanning. For example, a scientist chooses a set of temperatures that spans a phase transition in a sample. The experimental procedure then involves collection of a predefined amount of neutron data for each temperature set-point as neutron scattering experiments are often statistics limited, i.e. one has to record enough scattered neutrons in order to obtain useful results. Real-life experiments involve scanning of multiple variables, e.g. temperature, pressure, neutron polarization, etc.

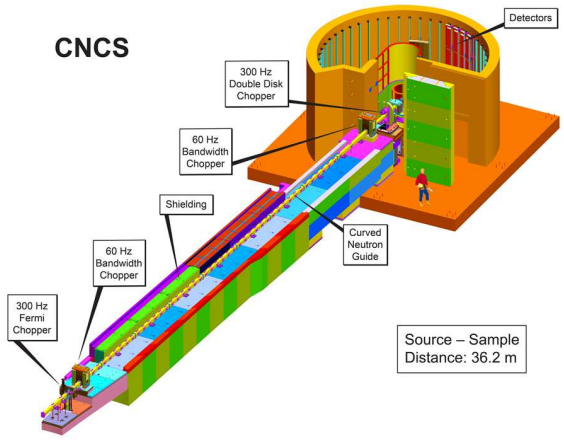


Figure 3. *Example neutron scattering instrument: Cold Neutron Chopper Spectrometer [CNC] at SNS.*

SNS Data Acquisition System

Such a complex instrumentation requires a very sophisticated data acquisition and control system that is usually custom designed and matches specifics of each neutron scattering facility. At SNS, the Data Acquisition System (DAS) [RIE] is a collection of real-time firmware running in FPGA-based boards, programs written in Microsoft Visual Studio C++ and National Instruments LabView [LAB] virtual instruments. Each piece of software typically performs one single task such as detector read-out or motor control and the programs communicate via TCP/IP networks. The SNS DAS is composed of four subsystems (see Figure 4):

- Real-Time System for detector electronics and read-out
- Timing System provides crucial timing information
- Slow Controls for sample environment, motors and ancillary equipment
- Control & Data for neutron data collection and main control

The Control Computer (see Figure 4) brings together information from each subsystem. It is designed to run a number of control programs that interact with its partners on the remote computers. For example motor control program communicates via TCP/IP with a motor satellite program.

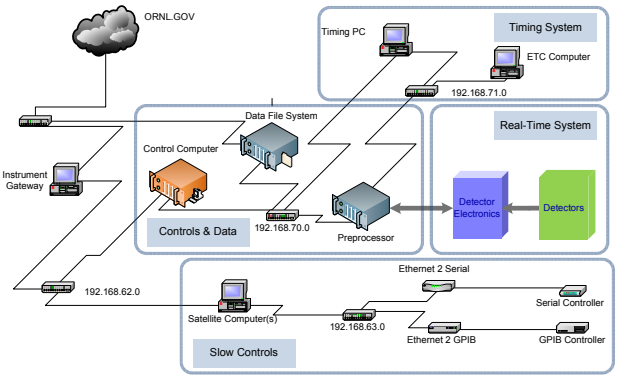


Figure 4. *SNS DAS Network Architecture*

PyDas - Python environment for SNS DAS

From the very start it was realized that the SNS DAS system needed a robust and convenient scripting environment. Python was chosen as a scripting language for its simplicity, elegance, power and ease to write extension modules. The SNS DAS scripting environment is called PyDas. It is designed to not only provide scripting capabilities but also the ability to integrate various experimental components into a single application.

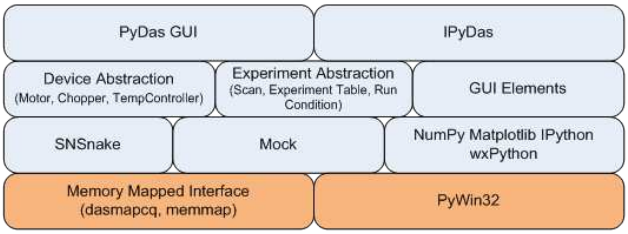


Figure 5. *PyDas General Architecture*

The general PyDas architecture is shown in Figure 5. Shared memory and Win32 API provide the interface to other DAS components. Two custom C++ modules (dasmapcq and memmap) and Mark Hammond's pywin32 [HAM] module are at the foundations of PyDas. Next layer contains SNSnake.py and mock.py modules that "pythonize" the low-level interface. The

former provides access to real hardware, while the latter delivers the essential simulation (mock) environment. PyDas also uses standard scientific Python modules such as NumPy [NUM], Matplotlib [MAT] and IPython [IPY]. The GUI environment is provided by wxPython [WXP]. Above these base layers there are modules that abstract hardware devices (such as motors, choppers and temperature controllers), experiment procedures (such as scan, experiment table, run condition) and various GUI elements.

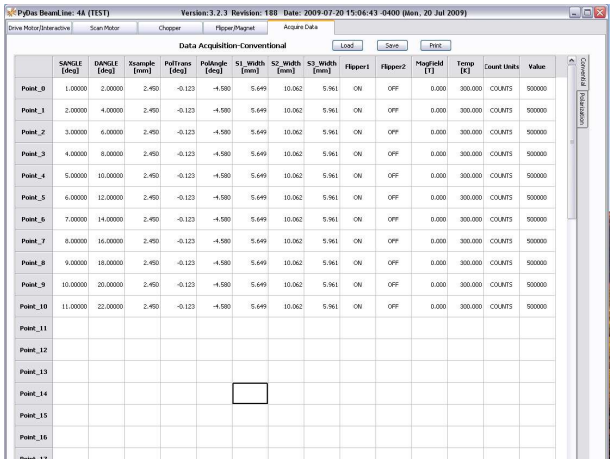


Figure 6. *PyDas GUI example*

PyDas can be run in two basic modes: with graphical user interface (GUI) and via the command line. The PyDas GUI interface (see Figure 6) is used to run predefined experiments. Each instrument is provided with a customized PyDas GUI to meet instrument individual needs. The command line interface, called IPyDas, is based on the IPython shell embedded into a single wxPython window and allows for custom experiment scripting. Below, we present two short examples of IPyDas scanning scripts:

```
scan_motor('motor1',arange(10.0,90.0,0.5),
  scan.RUNTIME,30.0,go_back=True)
scan_plot()


for temp in [273.0,293.0,313.0,333.0]:
    tempctrl.value = temp
    scan_chopper('Energy',[25.,35.,50.],
      scan.PCHARGE,1e10,
      title='The Nobel Prize Data at T=%s K' % temp)
```

Summary

In summary, we presented the complexities and opportunities of modern neutron scattering experiments and how PyDas helps to automate and integrate SNS Data Acquisition System. The future challenges for PyDas lie in the requirement to support 24 different instruments with high level of customization. Based on lessons learned so far we are in the process of refactoring PyDas to better meet these challenges, provide live neutron data viewing (currently served by a C++ Measurement Studio application) and we are also exploring new Python technologies such as IronPython [IRP].

The authors would like to thank Xiaodong Tao for initiating the PyDas GUI project and members of the SNS DAS Group: Lloyd Clonts, Gayle Green, Andre Parizzi, Mariano Ruiz-Rodriguez and Madhan Sundaram for collaboration and help during the development. ORNL/SNS is managed by UT-Battelle, LLC, for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[SQU] G.L. Squires, Introduction to the Theory of Thermal Neutron Scattering, Cambridge University Press (1978)

[PYN] R. Pynn, Neutron Scattering, A Primer, http://neutrons.ornl.gov/science/ns_primer.pdf

[ORN] ORNL Neutron Sciences, http://neutrons.ornl.gov

[KNO] G.F. Knoll, Radiation Detection and Measurement, J.Wiley & Sons (2000)

[CNC] Cold Neutron Chopper Spectrometer http://neutrons.ornl.gov/instrument_systems/beamline_05_cncs

[RIE] R.E. Riedel, Overview of Data Acquisition at the SNS, talk given at NOBUGS 2004 conference, http://lns00.psi.ch/nobugs2004

[LAB] National Instruments Labview, http://www.ni.com/labview

[HAM] M. Hammond, A. Robinson, *Python Programming On Win32*, O'Reilly Media (2000), http://pywin32.sourceforge.net

[NUM] T. Oliphant et al. NumPy, http://numpy.scipy.org/

[MAT] J.D. Hunter, *Matplotlib: A 2D graphics environment.* Computing in Science and Engineering. 9: 90-95 (2007). http://matplotlib.sourceforge.net.

[IPY] F. Perez and B. Granger: *IPython: a system for interactive scientific computing*, Computing in Science & Engineering 9(3) 21-29, 2007, http://ipython.scipy.org

[WXP] WxPython, http://www.wxpython.org/

[IRP] J. Huginin et al. IronPython, http://www.codeplex.com/IronPython

Progress Report: NumPy and SciPy Documentation in 2009

Joseph Harrington (jh@physics.ucf.edu) – *U. Central Florida, Orlando, Florida* USA
David Goldsmith (dgoldsmith_89@alumni.brown.edu) – *U. Central Florida, Orlando, Florida* USA

In the Spring of 2008, the SciPy Documentation Project began to write documentation for NumPy and SciPy. Starting from 8658 words, the NumPy reference pages now have over 110,000 words, producing an 884-page PDF document. These pages need to undergo both presentation and technical review. Our plans include SciPy documentation as well as several user guides, tutorials, and pamphlets. A critical need at this point is a stable funding stream to support the Documentation Coordinators who have been instrumental to the success of this project.

Abstract

In the Spring of 2008, the SciPy Documentation Project began to write documentation for NumPy and SciPy. Starting from 8658 words, the NumPy reference pages now have over 110,000 words, producing an 884-page PDF document. These pages need to undergo both presentation and technical review. Our plans include SciPy documentation as well as several user guides, tutorials, and pamphlets. A critical need at this point is a stable funding stream to support the Documentation Coordinators who have been instrumental to the success of this project.

Introduction

We maintain that, under any good development model, a software project's reference documentation is written approximately concurrently with code. In other words, the software includes full reference documentation as part of each release.

In our experience, many research codes do not follow such a model. Research codes typically start as solutions to specific needs and are written by the researchers who will use them. Documentation is often brief and incomplete, as the small circle of users is generally familiar with the code, having written it or having received training from those who did. The code may even be short enough and the users expert enough that reading the source code is a viable documentation method for them. Most such codes reach end-of-life when the original authors either move on or can no longer maintain them, or when the efforts for which they were written end.

However, some codes achieve a life of their own, with new people taking on maintenance and development in a community open-source effort. If the lack of documentation is not recognized and remedied at this stage, and if the developers do not require documentation to be written concurrently with any new code, the deferred documentation effort can become larger than any reasonable investment of the developers' time.

This was the case with NumPy. Between its forks, rewrites, and much organic growth, we are fortunate, in the authors' opinions, to have a single, stable package at all. It is perhaps not surprising that documentation development has not kept pace. Yet with thousands of users [Gen], some (perhaps many) of whom are learning to program for the first time using NumPy, we assert that documentation is important. As the first author outlined at the SciPy 2008 Conference [Har], he found himself, in the fall of 2007, teaching data analysis with NumPy without much documentation, and resolved that, by the fall of 2008, his class would have the documentation it needed. He then founded the SciPy Documentation Project (SDP). The following sections describe the SDP, its current state, and our future needs and plans.

The Project

Before the start of the SDP, documentation entered NumPy and SciPy as did any component of the code: through the Subversion (SVN) repository. Only code developers could enter documentation. Other community members rarely made contributions, and if they did, it was by emailing the developers or a relevant mailing list. It was clear from the volume of unwritten documentation that a much larger group would need to be involved. We needed approved documentation writers, reviewers, and proofers (the SDP's three labor catagories) to enter, edit, discuss, pass judgement on, and proofread the documentation. We also needed tools for organizing the effort and generating progress statistics. Thus was born the SciPy Documentation Wiki [Vir], now located at docs.scipy.org.

After independently but concurrently commencing work on versions of the Wiki, Pauli Virtanen and Emanuelle Guillart quickly combined efforts, with Virtanen taking on the long-term management of the Wiki and the tools behind it. Gael Varoquaux and Stefan van der Walt also contributed. The Wiki interfaces with the SVN repositories, allowing semi-automatic commits into the NumPy and SciPy sources. Fully automatic commits are not permitted because of the risk of malicious code introduction. If code developers modify documentation in the sources, that gets communicated to the Wiki so that it has the current version.

The reference documentation appears as the Python docstring [Goo] to each documented object in the NumPy sources. These ASCII documents are written

in a dialect of ReStructured Text [reST], which is quite readable without processing; the user sees the raw document in Python's help() facility. Prior to our taking the Wiki live, the community updated and refined the NumPy/SciPy Docstring Standard, to allow for formulae and figures. Because help() cannot display them, figures are discouraged in the NumPy documentation, but other packages adopting the template may permit them. The application Sphinx [Bran] produces HyperText Markup Language (HTML) and Portable Document Format (PDF) versions of the documentation, with indices. These processed forms are available at docs.scipy.org.

In addition to the public editing, commentary, and review system, there are instructions for participants on the front page, a recently-added Questions and Answers page, and a Milestones page that we use to organize and motivate the community, among other features. The system went live in the early Summer of 2008; since then, over 75 contributors have signed up for accounts. The Wiki generates weekly activity and participation statistics (http://docs.scipy.org/numpy/stats). Based on an examination of this page, we estimate that roughly 80% of the documentation has been written by about 13% of the contributors, but that most registered participants have contributed something.

Doc Coordinators

The SDP effort moves forward when the community works on it. To keep the community working, the first author has hired full-time Documentation Coordinators. Stefan van der Walt was the Coordinator in summer 2008, and continues his involvement in the project by handling documentation issues on the SciPy Steering Committee and working on issues related to SVN. The second author has been the Coordinator since June 2009. Among other duties, he has most recently been focused on updating the review protocol in anticipation of the NumPy doc review push (see below). As one can see from the Wiki's Status Page <http://docs.scipy.org/numpy/stats>'_, very little work was done between September 2008 and May 2009, during which period there was no Coordinator. (The fall 2009 hiatus is due to a delay in implementing a new dual-review protocol.)

The Coordinator keeps the community moving by defining milestones, defining short-term goals (like a weekly function category for everyone to work on), holding a weekly open teleconference with the writing community, and simply "talking it up" on the mailing lists to keep documentation on people's minds. The Coordinator also tends to write the pages nobody else wants to work on.

We also have motivators, in the form of T-shirts for those writing more than 1000 words or doing equivalent work such as reviewing, mention in NumPy's THANKS.txt file, and perhaps other incentives in the future. We are now soliciting contributions such as sponsored trips to the SciPy10 conference. Those interested in contributing sponsorships or ideas for incentives should contact the first author.

State of the Docs

The Wiki Status Page <http://docs.scipy.org/numpy/stats>'_ shows that there are over 2000 documentable functions, classes, and modules in NumPy, but that the sources contained just 8658 words of reference documentation when the SDP began, i.e., an average of a little more than four words per documentable object. Stefan van der Walt, the 2008 Documentation Coordinator, triaged the documentable objects into "Unimportant" and "Needs Editing" categories, and writing began.

Date	Needs Review	Being Edited	PDF Pages
Jun '08	6%	13%	<100
SciPy '08	24%	28%	309
SciPy '09	72%	6%	884

As of the SciPy '09 Conference (August 2009), the NumPy reference documentation consisted of over 110,000 words. Because the initial object triage was intentionally light, much of the final 15% of unedited objects are actually "Unimportant". This category mainly includes under-the-hood items that are never seen by normal users. With "submittable drafts" of the vast majority of reference pages, we will next focus on review.

Near Future: NumPy Review

The quality of the draft documentation is generally high, but writing is rarely complete without some critical review. The Project's original experiment with a one-review protocol resulted in very inconsistent documentation: some pages were approved by more technically-minded reviewers (such as developers), but had sections that were difficult to understand. Other approved pages, reviewed by professional writers, were missing important facts. We have thus defined and are implementing a two-review protocol, where both technical and presentation reviewers must approve each page. This requires changes to the Wiki (pending), changes to the reviewer instructions (finished), and recruitment of expert reviewers (we invite qualified readers to participate).

Next Projects

At the SciPy '09 Conference, the authors held a birds-of-a-feather (BoF) session where we discussed priorities for documentation after the NumPy and SciPy reference documentation is finished. The following ideas emerged:

NumPy User Guide: There are several pieces of text that a volunteer editor could integrate into a NumPy User Guide. The first are the concept pages in the numpy.doc module. These document NumPy concepts that occur in neither Python nor the most popular commercial scientific programming languages (e.g., Matlab or Interactive Data Language), such as broadcasting, indexing tricks, and other expediencies. There is a great deal of additional text in NumPy lead developer Travis Oliphant's original book, "Guide to NumPy." [Oli] Much of that work is technical documentation (C-API docs, etc.) or is appropriate for very expert users. This proposed User Guide would thus require much additional writing, but a great deal of excellent raw material already exists in the Wiki.

SciPyStack Test Drive (~10-pages): This document would narrate a demonstration that would give a first exposure to the power of NumPy. It would essentially be a marketing document, something one could give to colleagues to convince them it would be worth their time to learn more.

Getting Started Tutorial (~250 pages): This tutorial would teach a new user the basics of NumPy and some related packages, including array math, file I/O, basic plotting, some scientific methods packages, and where to learn more in all of these categories. The authors are aware of several such tutorials, but most are either unpublished or have been labeled by their authors as preliminary or unfinished. Others are outdated, having been written for one of NumPy's predecessor packages and not (yet) updated. Only a few are on the scipy.org web site. We may opt for one "General Tutorial" and several discipline-specific tutorials, such as the one for astronomy. [Gre]

Reading tools: The Python help() and NumPy np.info() functions are convenient, but are limited to ASCII text in the terminal. The numpy.info function should be enhanced so that, at the user's option, it can interface with external doc readers. For example, np.info(np.cos) might start a PDF reader and display the page that describes the np.cos function; or it might signal a Web browser to do the same using the HTML documentation.

SciPy Reference Pages: Documenting SciPy will be a monumental task. Though it starts from a much healthier state than did NumPy (at the time of article submission it had 628 pages of documentation, versus <100 for NumPy when the SDP began), the Doc Wiki shows that it has nearly twice as many documentable objects. Its content is highly technical, so writers with specialized knowledge will be required, arguably to a much higher degree than was needed for NumPy. Furthermore, much of the current documentation employs graduate-level technical language that many potential users may not understand. This work will begin in earnest once NumPy has been reviewed, but the SciPy portion of the Wiki is open, so those who are motivated, and in particular module maintainers, may begin this work at their convenience.

SciPyStack User Guide: This would be an integrative document to teach the fundamentals of the entire toolstack. We see it as covering basic (I)Python, giving pointers to introductory books on Python, teaching array basics (essentially the current NumPy User Guide content), presenting file I/O (including HDF, FITS, etc.), and having chapters on numerical computation, visualization, parallel computing, etc. To produce it, we propose following the academic "edited book" model. We would have a community discussion to agree on a table of contents and book conventions. A Request For Proposals would then solicit author teams to write chapters. An editor (or editors) would assemble and integrate the chapters, attending to things like style uniformity, and cross references, and indexing. We see as the final products a free e-book and a paper edition for retail sale. Our vision includes maintaining it to stay current.

What to Call It?

Readers will notice use of the term "SciPyStack" above. The community thinks of "SciPy" in two contexts: as a specific package and as the loosely defined collection (or stack) of packages used together for scientific computing in Python. The dual use causes confusion. For example, the authors have been involved in many discussions where someone strongly objected to a proposal, thinking that it was to add a major capability to the package when it was instead to produce a freestanding capability of which the unaltered package would be a component. As the community works on improving both the package and the stack, and as documentation moves to a stage where we are writing integrative material that must address both concepts together, it is our opinion that having separate words for the concepts will be beneficial. We offer here some thoughts and a proposal to initiate discussion.

Capitalization ("scipy" for the package vs. "SciPy" for the stack) has been insufficient, and few would wish to say "SciPy the stack" and "scipy the package" on each use. To avoid confusion, one of these usages much change. Changing the name of the package would shatter its API. We also have a web site, scipy.org, whose widely-known name we would be foolish to change, yet it is the portal to the stack. Our name for the stack would thus best begin with the fragment "SciPy..." for consistency with the well-known site.

We considered many possible names, including "Scientific Python" (already taken by another package) and "Py4Sci" (disagrees with web site name), and "SciPython" (likely to revert to "SciPy" in casual speech). "SciPyStack" could work; it is explicit, not easily abbreviated back to "SciPy", and short. Others might work as well. We raise the issue now to encourage discussion before writing begins on integrative documentation.

Conclusions

The SDP's first project has been reference documentation for NumPy. The vast majority of reference pages for user-visible objects are now ready for review, and these drafts have already been included in several NumPy releases. Given that most objects lacked documentation prior to the project, we feel these drafts are a significant improvement over the prior state, and the testimonials of students in courses taught by the first author bear out the notion that NumPy is much easier to learn today than it was two years ago. Once the NumPy documentation is finished, we will move on to SciPy. Future plans also include several books and a pamphlet. However, the big labor requirement now is for technical and presentation reviewers for the NumPy reference documentation.

There is another need, however: a sustainable funding stream to pay a Documentation Coordinator (which is all but essential for continued progress) and perhaps others in the future, such as book editors. The first author has been paying the Coordinators from his research grants, with the justification that this is ultimately costing the grants less than paying his group for the time they would require to learn NumPy without documentation. This cannot continue much longer, but the return to the community has already been significant. We believe that NumPy represents "bread and butter" to thousands of technical professionals worldwide, and that the number is quickly increasing. This software is not free of cost, only free of charge, and its continued development depends on contributions. We feel strongly that those who benefit substantially from NumPy, and especially those who profit from it, should consider contributing labor or funds. We are pursuing both commercial and governmental funding opportunities to continue supporting the SDP, and we seek senior collaborators for proposals. We invite those interested to contact us so that we may all continue to benefit from the power of the SciPy software stack.

Acknowledgements

Support for this work was provided by NASA through an award issued by JPL/Caltech.

References

[Bran] Georg Brandl, Sphinx: Python Documentation Generator. http://sphinx.pocoo.org/, 2009.

[Gen] Igor Genibel, Christoph Berg, Ian Lynagh (graphs), et al. Debian Quality Assurance: Popularity contest statistics for python-numpy. http://qa.debian.org/popcon.php?package=python-numpy, October, 2009.

[Goo] David Goodger and Guido van Rossum, Python Enhancement Proposal 257: Docstring Conventions. http://www.python.org/dev/peps/pep-0257/, 2001-9.

[Gre] Perry Greenfield and Robert Jedrzejewski, Using Python for Interactive Data Analysis. http://stsdas.stsci.edu/perry/pydatatut.pdf, 2007.

[Har] Joseph Harrington, The SciPy Documentation Project. In Proceedings of the 7th Python in Science Conference, G. Varoquaux, T. Vaught, and J. Millman (Eds.), pp. 33 – 35, http://conference.scipy.org/proceedings/SciPy2008/paper_7/full_text.pdf, 2008.

[Oli] Travis Oliphant, Guide to NumPy. http://www.tramy.us/numpybook.pdf, 2006.

[reST] reStructuredText: Markup Syntax and Parser Component of Docutils. http://docutils.sourceforge.net/rst.html, 2006.

[Vir] Pauli Virtanen, Emmanuelle Gouillart, Gael Varoquaux, and Stefan van der Walt, et al. pydocweb: A tool for collaboratively documenting Python modules via the web. http://code.google.com/p/pydocweb/, 2008-9.

SciPy 2009 conference

Python for Scientific Computing

Proceedings of the 8th Python in Science Conference
SciPy Conference – Pasadena, CA, August 18-23, 2009.
Editors: Gaël Varoquaux, Stéfan van der Walt, K. Jarrod Millman

ISBN 978-0-557-23212-3